AFIT/GCS/ENG/99M-19

Generating Executable Code from Formal Specifications of Primitive Objects

THESIS

Travis W. Tankersley
1Lt, USAF

AFIT/GCS/ENG/99M-19

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

AFIT/GCS/ENG/99M-19

Generating Executable Code from Formal Specifications of Primitive Objects

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Travis W. Tankersley, B.S. Computer Science

1Lt, USAF

March, 1999

AFIT/GCS/ENG/99M-19

Generating Executable Code from Formal Specifications of Primitive Objects

Travis W. Tankersley, B.S. Computer Science

1Lt, USAF

Approved:

Robert P. Graham	5 Mar 99
Maj. Robert P. Graham Jr. Committee Chair	Date
Thomas C. Hartrum	5 Mar 99
Dr. Thomas C. Hartrum Committee Member	Date
Scott A. DeLoach	5 Mar 99
Maj. Scott A. DeLoach Committee Member	Date

Acknowledgements

There are many people who enabled this thesis effort to become a reality. I am extremely grateful to Major Graham, my thesis advisor, who always answered my questions with more detailed questions forcing me to continue to discover areas of my research that needed more effort. Dr. Hartrum and Major Deloach, my other committee members, were always available for discussions and advice about my research when I needed a different viewpoint. Also, I would like to thank Major Schorsch for taking time out of his own research to provide insight and training on a multitude of subject areas.

I am grateful to Lt. John Kissack for being there to bounce ideas and problems off of and always offering a quick wit when things got a bit too serious in the lab. My family, especially my wife Michelle and son Trent were ever supportive when I desperately needed it. Without their support I could have never maintained my sanity throughout this process.

Travis W. Tankersley

Table of Contents

List of Figures

List of Tables

Abstract

The concept of developing a model for producing compilable and executable code from formal software specifications has long been a goal of software engineers. Previous research at the Air Force Institute of Technology (AFIT) has been focused on specification and domain analysis. An analysis model is populated using specifications written in *Z*. Then, a set of preliminary design transforms refines the specification in the analysis model. This research bridges the gap between analysis and design, allowing source code to be produced from formal specifications of primitive objects using transformational programming. The contribution of this thesis is to transform the analysis model for primitive objects into a design model representing the primitive objects, and to produce compilable and executable source code in Ada 95 from the resulting design model.

Generating Executable Code from Formal Specifications

of Primitive Objects

I. Introduction

The concept of developing a model for producing compilable and executable code from formal software specifications has long been a goal of software engineers. The reasons this concept is so appealing as an alternative to standard software development practices revolve around the fact that during development, specifications continuously evolve towards the ultimate final product needed by the end user. However, the development of the same code rarely, if ever, is capable of staying in step with the specification evolution and as a result additional releases are required to incorporate the changes. There are many downsides to this approach, but the two most significant problems are first, the original version of the software provided to the end user is rarely the package necessary for performance of day-to-day operations and second, the amount of time required to produce the next iteration of the software far exceeds the deadlines specified by the end user. With the production of software from specifications, the only part of the project that must maintain agreement with the end user is the specification. Generation of the next version of software then becomes automatic. This thesis is a study and proof of concept that this approach can be realized for primitive object specifications (i.e., objects without aggregation) in the specification. There are some limitations on the proof of concept that will be addressed. Research previously accomplished at the Air Force Institute of Technology (AFIT) produced a domain model from a formal specification written in *Z*. The contribution of this thesis is to transform the domain model for primitive objects into a design model for those same primitive objects and from the design model to produce Ada code.

1.1 Background

This research began with a system in place to produce a domain model from formal specifications. This system is referred to as *AFIT*tool and has been a continuing project

at AFIT for several years. During the same time this research was being done, related research for eliciting and harvesting design decisions from the designer based upon information found in the domain model [1] was also being done, as well as research for handling aggregate objects found in the domain model [4]. As a result of the latter of these two research projects, a target design model used to represent the specification had to be defined and agreed upon. The design model built was based upon research done as part of Sward's PhD dissertation [14] for reverse engineering Fortran code into a generic object-based model. The model Sward built for storing the reverse engineered Fortran provided a good basis for design. Some modifications were required, however, and the final model is described in detail in Chapters III and IV. Figure 1 shows the overall picture of *AFIT*tool outlining the significant areas addressed during this effort.

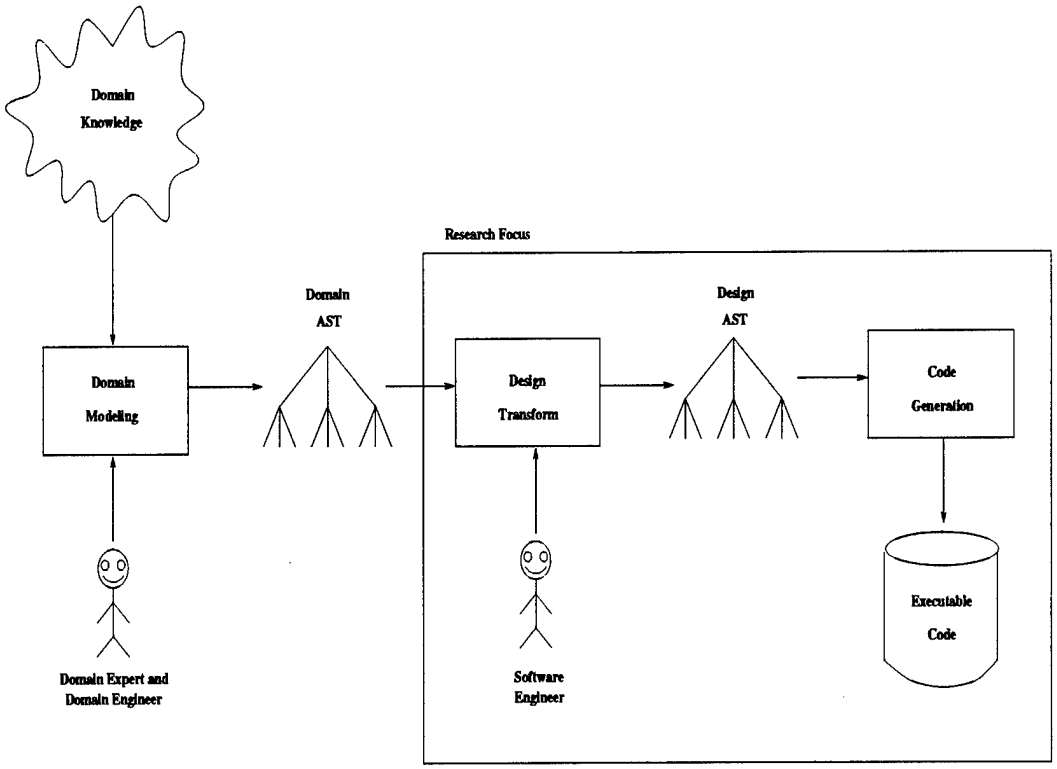


Figure 1 *AFIT*tool Background and Research Focus

A series of transformations was developed to produce the design model from the domain model. The most intricate of these transformations deal with transforming various *Z* predicates and expressions into imperative and object-oriented programming constructs.

The organization and definition of the *Z* predicates addressed by this research is discussed at length in Chapter II. The explanation of their transformation into program constructs can be found in Chapter IV.

1.2 Problem Statement

The goal of this research is to demonstrate the ability to produce a design model based upon the information found in the domain model for primitive objects, and produce code from the design model using Knowledge Based Software Engineering (KBSE) tools.

1.3 Approach to Problem

The approach taken in this thesis is one of incremental transformations of the user's original problem specification. The idea behind transformational programming is to modify a program by applying rules. These rules attempt to match some portion of the program with a test condition. When this condition is met, the rule fires and this portion of the program is converted to, or replaced by, a different but equivalent structure in the program [5]. The first series of transformations takes place within the domain model to prepare it for transformation to the design model. The next series of transformations produces a separate design model based on the information found in the domain model. This transformation does not produce a complete design, however. All methods of the domain model are initially transformed into procedural methods in the design, and the predicates and expressions present in the domain model are mapped to their corresponding methods in the design model but are not transformed to an imperative or object-oriented format. Another set of transformations is then required to determine which methods should become functional methods versus procedural methods, and each predicate and expression is then transformed into its appropriate representation in the design model. Once this series of transformations has taken place, the design model is suited to produce Ada code using output grammars designed for this purpose.

1.4 *Contributions*

This thesis implemented a solution capable of generating source code from formal specifications using a limited set of predicates and expressions. It provides a framework for a complete transformation system from specification to code. The implications of this research indicate that through extensions to this system, a complete transformation system for generating code from specifications can be realized.

1.5 *Overview of Remainder of Thesis*

This thesis consists of five chapters. Chapter II takes a look at the literature reviewed prior to and during the thesis effort in order to validate the manner in which the thesis was approached. Chapter III explains the methodology or approach to the problem in greater detail to include a brief synopsis of the transformations needed to implement this solution. Chapter IV describes the details of the implementation. Finally, Chapter V discusses the findings of my research along with a brief description of future research that will be necessary for a complete working model. For a greater understanding of Z and Z notation, the reader is directed to Spivey [13].

II. Literature Review

2.1 Overview

The basis for this thesis effort was designed at the Air Force Institute of Technology (AFIT) by Masters students and is known as *AFIT*tool. It is built using the parsers, transformational qualities, and abstract syntax tree structures provided by the Refine language [8]. The Refine language supports the transformational programming method covered in Chapter I. The basic premise behind the development of *AFIT*tool is that only the specification should be maintained and when changed, the source code can be re-created using the tool. *AFIT*tool requires minimal interaction with the user and as such can be used by anyone familiar with the *Z* specification language.

This thesis effort is also based upon a second system developed at AFIT, a reverse engineering tool that was developed to convert imperative code into canonical object-oriented code. From this canonical representation, object-oriented source code can be produced. The design model developed for this research is based on Sward's canonical model [14]. Thus, the vast majority of this literature review focuses on literature generated at AFIT regarding the current status of these two tools, their strengths and weaknesses. Finally, the methodology underlying *AFIT*tool is based on Rumbaugh's Object Modeling Technique (OMT). This chapter provides the reader with the basic background of OMT necessary to understand this effort; for more information, see [11].

At the start of this research, *AFIT*tool contained most of the necessary features to handle the transformation from *Z* specifications into an object-oriented domain model represented as an Abstract Syntax Tree (AST) in Refine [8]. However, there are shortfalls that will be described later in this review. These shortfalls prevent the domain model from being completely and accurately captured in a format that can be transformed to the generic object-oriented model that represents the design model. The design model also had various shortfalls. These shortfalls prevented some portions of the domain model from being represented in the design model. The literature reviewed provides an insight into both models. As both were built without regard to how the other functions, the transformation process from one model to the other required a significant effort.

2.2 Related Work

The concept of design tools producing executable source code from specifications is not new. There have been several programs developed to aid in this effort but to date none are considered robust. For instance, Rational Rose is an automated program design tool that provides an interface allowing the user to represent a specification using pictures. The system then analyzes the pictures and produces source code for the user. One of the primary drawbacks to Rational Rose is that it does not handle the dynamic portion of the object model dealing with states, transitions and events. Neither does it allow the user to define or generate bodies for methods.

Another tool developed for part of this process is known as the Kestrel Interactive Development System (KIDS). Using KIDS, a developer specifies a program in terms of its pre- and post-conditions, then applies a semi-automated "design-tactic" to specialize a generic algorithm to a particular problem. Further refinements can then be applied to produce a more efficient implementation of the algorithm [5]. The major drawback to KIDS is the level of maturity needed by the designer for applying refinements in order to produce the most efficient algorithm.

CIP (Computer-Aided Intuition-Guided Programming) is a transformational programming tool that transforms abstract specifications to concrete programs. It uses a wide-spectrum-language that encompasses all constructs from the high-level specification to the functional kernel. By applying a series of correctness preserving transformations to the specification, CIP produces a verifiably correct concrete program equivalent to the original abstract specification. This is the approach that should be used when extreme software reliability is required. However, when extreme software reliability is not an issue CIP can be manually guided to transform an existing concrete program. The drawback to CIP is the amount of time spent building the abstract specification. The cost-effectiveness of deriving programs from abstract specifications is much less than that of developing a concrete program and performing the manual transformations. Therefore, abstract specifications are only recommended when extreme software reliability is an issue [5].

REFINE is an executable specification language used to build forward and re-engineering tools. It is the product of program synthesis research [5]. Currently, commercial systems are available to re-engineer existing source code to different platforms and to attempt to validate Y2K compliance. The REFINE language uses ASTs to represent the structure of a program. It then analyzes the AST with rules performing transformations when the antecedent of a rule matches a pattern in the AST. The commercially-provided systems are language specific and many languages are not supported. REFINE can also be used to develop transformational systems and was used for this effort.

Other tools exist that aid in program derivation and design. For a more thorough review of these tools, the reader is directed to Lowry [5].

2.3 Domain Model

In order to understand how the domain model relates to primitive object classes, it is necessary to break the domain model into components. These components are the object model, the dynamic model and the functional model. Combined, these models represent the Object Modeling Technique (OMT) as described by Rumbaugh [11].

2.3.1 Object Model. The fundamental OMT component, the object model, is the most important model in that it provides the template for developing an object-oriented system. It captures the static structure of the system through the use of objects. By building the system around objects, a real-world representation of an application is developed that patterns current practice more completely and accurately. The purpose of the object model is to describe objects. Rumbaugh states "an object is simply something that makes sense in an application context" [11]. The advantages of using objects are twofold: by breaking a problem down into objects that represent a physical entity, it is easier both to 1) understand and 2) implement.

An object class, or class, describes groups of objects that are similar in properties, semantics and behavior. The objects in a class will have the same attributes. This allows for common definitions and operations to be written once and re-used by all objects in the class. In addition, a class can inherit from another class where extensions can be made

and methods can be specialized to meet the needs of the sub-class. The object model also includes associations and aggregation relationships, but since this research effort focuses on primitive objects only, they will not be described. Figure 2 shows the implementation of the domain model as it relates to the class structure.

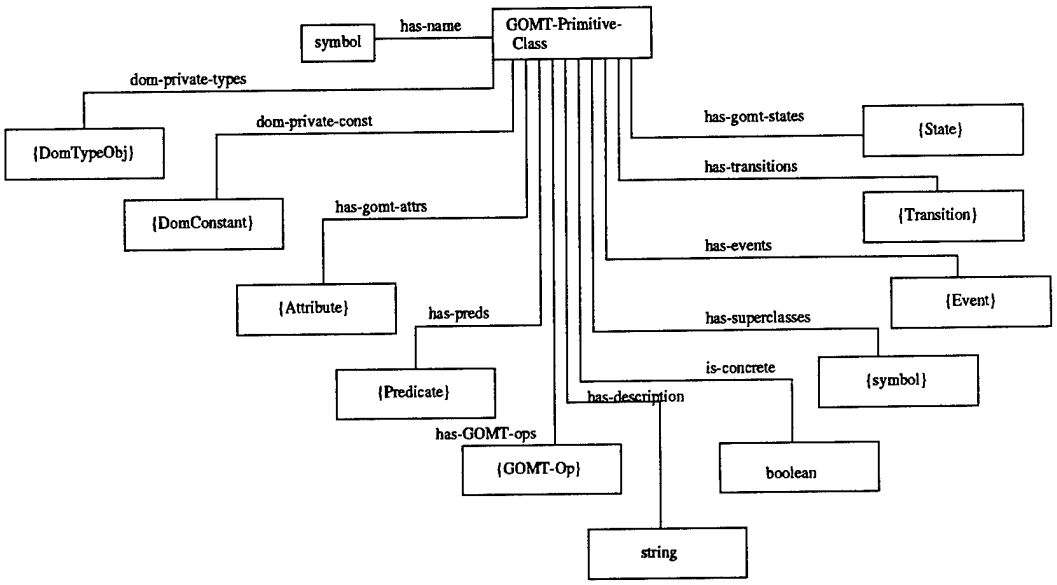


Figure 2 Domain AST for Primitive Class

The tree attributes *has-gomt-states*, *has-events* and *has-transitions* are part of the dynamic model. *Has-GOMT-Ops* represents part of the functional model [3]. Inheritance is supported through the use of *has-parents* and *is-concrete*. The AST permits multiple inheritance, but the transformations developed in this research effort support only single inheritance. That is, a class can only inherit from a single super-class.

2.3.2 Dynamic Model. The behavior of an object is captured in the dynamic and functional models. The aspects of a system that are related to time and changes are found in the dynamic model. Control of a system is defined as the "sequences of operations that occur in response to external stimuli" [11]. The dynamic model is composed of states that the object can be in based on the values of the attributes associated with the object. An object transitions from one state to the next as a result of an event. Events represent some external stimulus on an object. The pattern of events, states and transitions is modeled in a state transition table. To see the relationship of the dynamic model with the domain

model, notice in Figure 3 that a GOMT-DomainTheory is composed of a set of classes. The GOMT-DomainTheory meta-model as defined in the AST is shown in Figure 3.

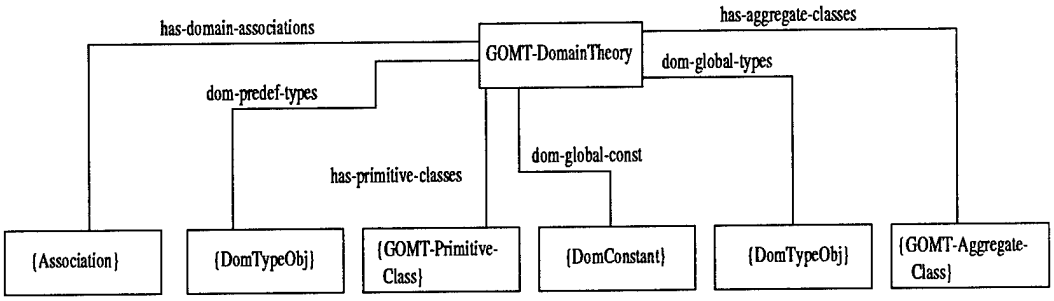


Figure 3 Top Level Domain Structure Depiction

The AST for the dynamic model defines states, transitions and events that represent the information listed in a state transition table. States, transitions and events are broken down further in this model as shown in Figure 4.

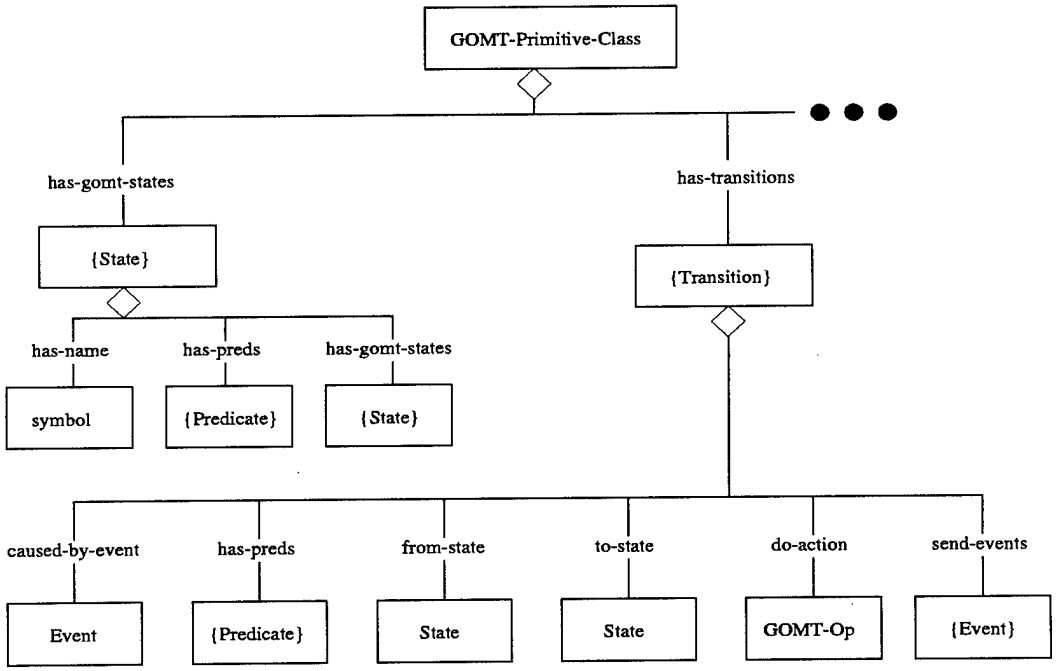


Figure 2.3. Class AST for Dynamic Model

Figure 4 AST for Dynamic Model

The representation of states, transitions and events in the domain AST does not directly lend itself towards an implementation. These pieces of the dynamic model must

be refined into methods that implement the desired behavior. After transformation into methods occurs, they then become part of the functional model.

2.3.3 Functional Model. The functional model captures the computational portion of an object's behavior [3]. It defines the set of methods or operations that can affect the class's attributes, and during design it incorporates methods that implement the object's dynamic behavior as well. The functional model is based on the concept of functional decomposition. This consists of decomposing an operation into a set of operations, where an operation with no sub-operations is considered a leaf node. Leaf nodes represent methods and comprise the functional model for a class and fit into the domain model as shown in Figure 5.

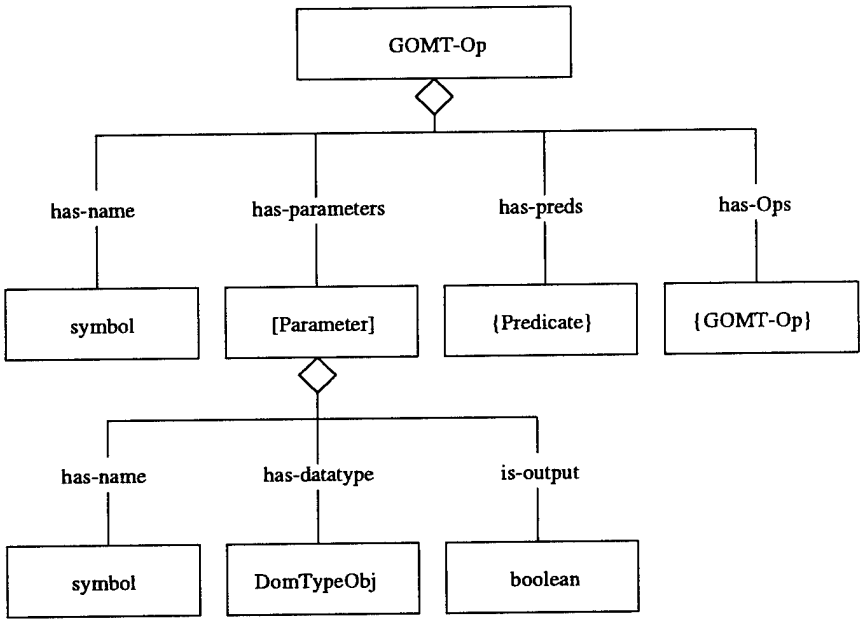


Figure 5 Domain Functional AST

A GOMT-Op can be composed of a set of GOMT-Op(s) allowing for the functional decomposition mentioned earlier. The *is-output* attribute allows a boolean value to be set for output parameters of a GOMT-Op. Any parameter that is not output is considered input to the operation. Most programming languages support parameters as input, output or input-output, so this is an area that must be addressed further along in the design of the system.

The implementation and meaning of the object, dynamic and functional models have now been described using the domain model AST in *AFIT*tool. It was somewhat stable, but still needed work to prepare it for transformation to a separate model used for design. This additional preparation constituted a large portion of this research effort.

2.4 Z Predicate and Expression Definitions

The *Z* predicate structures in the domain model are components of a GOMT-Op as shown in Figure 5. They serve as pre- and post-conditions of each operation found in the domain model. The post-conditions of each operation are transformed to code, but it is not the intent of this thesis effort to provide a transformation for every available predicate and expression found in a *Z* specification. Algorithm design is arbitrarily hard and far beyond the scope of this effort. The intent was to address those predicates and expressions that are commonly found in order to produce a proof of concept that could readily be used and extended incrementally to incorporate the addition of predicates and expressions. Each predicate and expression is represented in an AST format. The predicates and expressions transformed in this thesis effort are discussed and shown as they appear in the AST for the domain model.

2.4.1 True and False Predicate. The purpose of the true and false predicate as used in a *Z* specification are to represent the boolean values True and False. The structure of these two predicates are identical and represented in Figure 6.

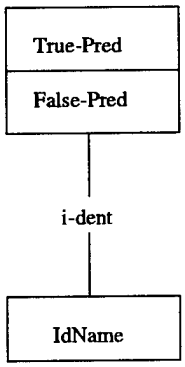


Figure 6 True and False Predicate Structure

2.4.2 Negated Predicate. The purpose of the negated predicate is to perform the task of reversing the boolean value returned by any predicate. The structure of the negated predicate is represented in Figure 7.

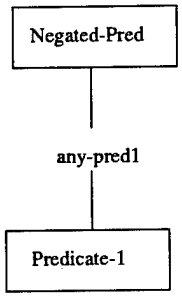


Figure 7 Negated Predicate Structure

2.4.3 Conjunct, Disjunct, Equivalent and Implication Predicate. The structure of all four predicate types are equivalent and therefore listed together. The purpose of the conjunct predicate is to provide the boolean 'and' capacity. The purpose of the disjunct predicate is likewise to provide the boolean 'or' capacity. The purpose of the equivalent predicate is to provide the boolean 'if and only if' capacity. Finally, the purpose of the implication predicate is to provide the boolean 'implies' capacity. The structure of this group of predicates is represented in Figure 8.

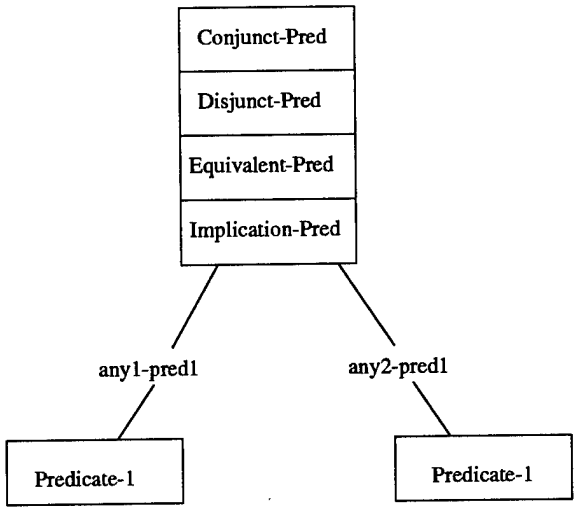


Figure 8 Structure for Conjunct, Disjunct, Equivalent and Implication Predicate

Note that while most target languages would support the boolean 'and' and 'or', many languages do not support the boolean 'if and only if' and 'implies'. Therefore these structures would not be found in the design model and would have to be transformed into functionally equivalent and correct structures for an imperative program.

2.4.4 Schemaref Predicate. The purpose of the schemaref predicate is to enable a predicate to reference some other object in the domain. This object could be an attribute, variable, parameter or a constant. The structure of the schemaref predicate is represented in Figure 9.

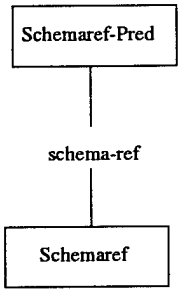


Figure 9 Schemaref Predicate Structure

2.4.5 Relational1 Predicate. The purpose of the relational1 predicate is to provide a means for testing the relationship between two expressions based upon the relational symbol used between them. This research concentrated on the equality, greater-than, greater-than-or-equal, less-than, less-than-or-equal, and the not-equality relational symbols. The structure of the relational1 predicate is represented in Figure 10.

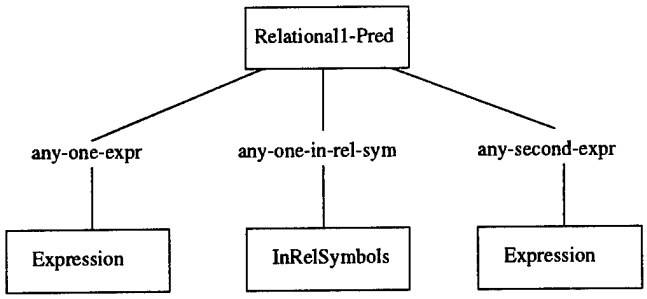


Figure 10 Relational1 Predicate Structure

The domain model also defines a relational2 predicate that uses two relational symbols between three expressions (i.e. A > B > C) as well as a relational3 predicate that uses three relational symbols between four expressions in the same manner. This effort only deals with the relational1 predicate as the use of relational2 and relational3 predicates is discouraged. Any predicate represented using a relational2 or relational3 predicate construct can be represented equivalently by conjuncting two or more relational1 predicates.

2.4.6 Bracket Predicate. The purpose of the bracket predicate is to enable predicates to be grouped or parenthesized to ensure proper execution when standard order of execution is not wanted. The structure of the bracket predicate is shown in Figure 11.

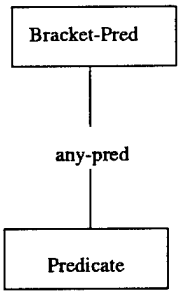


Figure 11 Bracket Predicate Structure

2.4.7 Variable Name Expression. The purpose of the variable name expression is to enable an object in the domain to be referenced in an expression. This object could be an attribute, variable, constant, or parameter as discussed in the schemaref predicate. This ambiguity of object type leads to a lengthy process of transformation and as a result becomes a search of the design tree to locate and identify the actual object being referenced. The structure of the variable name expression is represented in Figure 12.

2.4.8 Addition, Subtraction, Modulo, Multiplication and Division Expressions. The structure of all four expression types are equivalent and therefore listed together. The purpose of these expressions is to provide the mathematical capabilities indicated by each of their names. The structure of this group of expressions is represented in Figure 13.

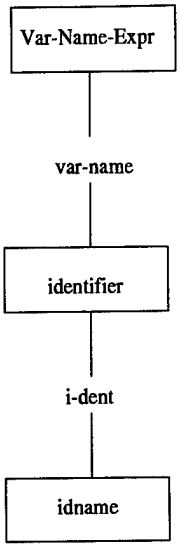


Figure 12 Variable Name Predicate Structure

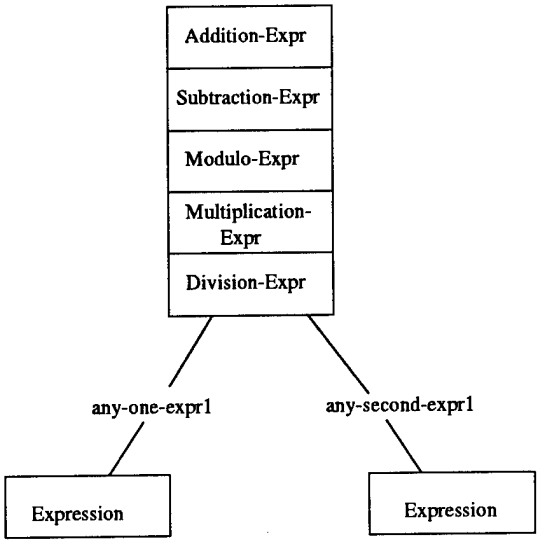


Figure 13 Structure for Addition, Subtraction, Modulo, Multiplication and Division Expressions

2.4.9 Exponent Expression. The purpose of the exponent expression is to provide the capacity to raise an expression value to the exponent represented by another expression. The structure of the exponent expression is shown in Figure 14.

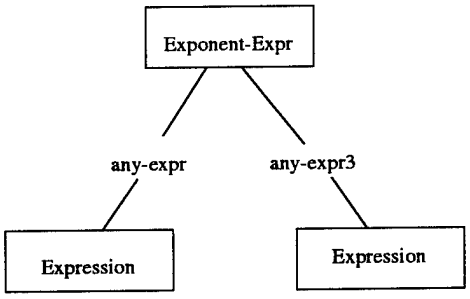


Figure 14 Exponent Expression Structure

2.4.10 Integer and Real Expressions. The purpose of the integer and real expressions is to provide the capacity for representing numbers in either integer or real format. The actual number is stored in the expression, therefore no representation of the structure is necessary for this expression type.

2.5 Design Model

Next, a design platform for representing the specification was needed. Several options were considered at this juncture, such as 1) converting the domain model in place to produce a wide spectrum language representative of both the design and domain model and 2) building a design model from scratch. The third and most promising approach to this problem was to use an existing model that was developed for reverse engineering and with minor modifications produce a design model that would meet the needs of this effort. This yields two narrow spectrum languages with one representing the specification in the domain model and the other in the design model. The foundation of such a model was found in Sward's PhD dissertation [14]. The model he used was called the Generic Object Model (GOM). Its original implementation is described in Chapter III and its implementation as used for this research effort is described in Chapter IV.

2.6 Conclusion

There are some commercial systems available for producing source code from specifications. However, each system has one or more drawbacks making it less than optimal. This review studied in-depth a partial solution producing an analysis model representing a specification written in *Z*. The analysis model is complete in that it covers the aspects of the object, functional and dynamic models described by Rumbaugh's OMT [11]. *Z* specifications and predicates are used to describe the valid operations on a class structure. This review also mentioned a target design model to be used for transforming the specification from analysis to design. Using these components, a transformation system can be implemented covering each portion of the OMT.

III. Methodology

3.1 Overview

One decision of this research was to use the tools in place for producing a domain model from formal specifications as much as possible. Several of the transformations needed to prepare the domain model for translation into a design model were in place; however, extensive modifications were required in order to implement inheritance during the transformations. The addition of transformations and their functionality is discussed later in Section 3.2. This research provided a design model capable of representing the information from the domain model in a design format. The need for a design model required extensive research into possible avenues of 1) expanding the domain model into a design model, 2) building a design model specifically for this task, or 3) finding a suitable model that could be used as is, or with minor modification. The third avenue seemed the most desired approach, but finding and approving a model took considerable effort. With the requirements for a design model determined, analysis and understanding of available models was necessary. The product of this analysis produced the Generic Object Model (GOM) designed by Maj Rick Sward [14]. It did, however, require some modification in order to produce the canonical model desired for this effort. Given these two pieces, a series of transformations were designed and implemented. The first added the concept of inheritance to transformations already present in the domain model, and one new transformation. The next set of transformations then converted the domain model to the design model. Many of the structures from the domain model were converted into new structures of the design model; however, the predicate structure of the domain model is very complex and as a result was simply copied into the design model. The next series of transformations converted predicates and expressions into design model constructs. Finally, two grammars were produced to generate Ada specifications and bodies from the domain model.

3.2 Extending Transformations to Handle Inheritance

Prior to this research effort there were several existing transformations that modified the domain model; these transformations prepare the domain model for conversion to

a design model. The first of these transformations deals with converting an invariant constraint on a single existing attribute into a type definition in the domain model and did not require any modification during this thesis effort. The second transformation is responsible for the creation of a *Get_Attribute* and a *Set_Attribute* method for each attribute associated with a class in the domain model. Again, no modifications were necessary for this transformation during this thesis effort.

The third transformation deals with derived attributes involved in an equality predicate. The purpose of this transformation is to modify the operations in the domain model so they would update the derived attribute each time one of the attributes it was derived from was updated. The transformation as written worked fine when all of the attributes involved in the transformation were local to the class, that is, they were not inherited from a super-class. When attributes were inherited, the transformation as written could not locate the attribute in question and would cause an abrupt termination of the transformation process. While one approach would be to simply not allow this transformation to take place if inherited attributes were involved, it seemed feasible that this situation could be handled with some effort. This transformation was modified to produce a polymorphic method at the sub-class level that included the invariant constraint of the equality predicate as a post-condition.

The fourth transformation also deals with derived attributes involved in an equality predicate. The purpose of this transformation is to remove the attribute and the *Set_Attribute* method from the class for the derived attribute and to modify the *Get_Attribute* method to calculate the value of the attribute each time it is needed. Again, this transformation works fine as long as none of the attributes involved in the equality predicate is inherited from another class; however, this is not always the case. The methodology decided upon to address inheritance during this transformation is to produce a polymorphic method at the sub-class level that included the invariant constraint of the equality predicate as a post-condition. However, if the derived attribute was inherited, an informational message was generated and sent to the user indicating that the third transformation should be chosen instead.

The fifth transformation again deals with derived attributes involved in an equality predicate. The purpose of this transformation is much different than either of the previous two. It is intended to *Set_All_Attributes* with a single method while removing all of the individual *Set_Attribute* methods for the attributes involved in the predicate. The methodology used to implement this transformation was to build the new *Set_All_Attributes* method and then remove the individual *Set_Attribute* methods from the local class in the domain model. When attributes in the predicate are inherited from a super-class, this transformation provides a null operation for each inherited *Set_Attribute* method at the local class.

The sixth transformation was the final transformation dealing with the derived attributes in an equality predicate. The purpose of this transformation is similar to the fifth transformation. Its intent is to build a *Set_Some_Attributes* method where the attributes involved in the new method would be members of the equality predicate. When the attributes are local, the transformation builds a new *Set_Some_Attributes* method and removes the individual *Set_Attribute* methods for the attributes involved in the predicate. In principal several constraints should have been placed on the operation of this transformation. For instance, a minimum of two attributes should have been required in each of the *Set_Some_Attributes* methods, and the transformation should have continued to cycle until all of the attributes involved in the equality constraint had been included in at least one *Set_Some_Attributes* method. In practice, these constraints were never forced upon the transformation and were considered part of the necessary repairs. When attributes in the predicate are inherited from a super-class, this transformation provides a null operation for the *Set_Attribute* method at the local class.

The seventh transformation deals with inequality predicates. This transform is described by Hartrum in [2] but never implemented. The purpose of this transformation is to recognize that an inequality invariant constraint exists, and then add a copy of the predicate as a post-condition to all of the *Set_Attribute* methods involved in the predicate. This indicates that the constraint should hold after the execution of the *Set_Attribute* method. The methodology decided upon for this transform was to add the predicate to the *Set_Attribute* method if the attribute were local, but if the attribute was inherited the transform would search the inheritance tree until the appropriate attribute was found, copy

the *Set_Attribute* method from the super-class to the local class in the domain model, and finally add the inequality predicate as a post condition to the *Set_Attribute* method. This methodology meets the intent of the original concepts of the transformation and provides a strategy for implementing the transformation.

3.3 Design Model

The domain model was prepared for transformation to the design model using the seven previously mentioned transformations. Another series of transformations was built to populate the design model from the domain model, and another set of transformations then prepared the design model to produce source code. This process is depicted in Figure 15.

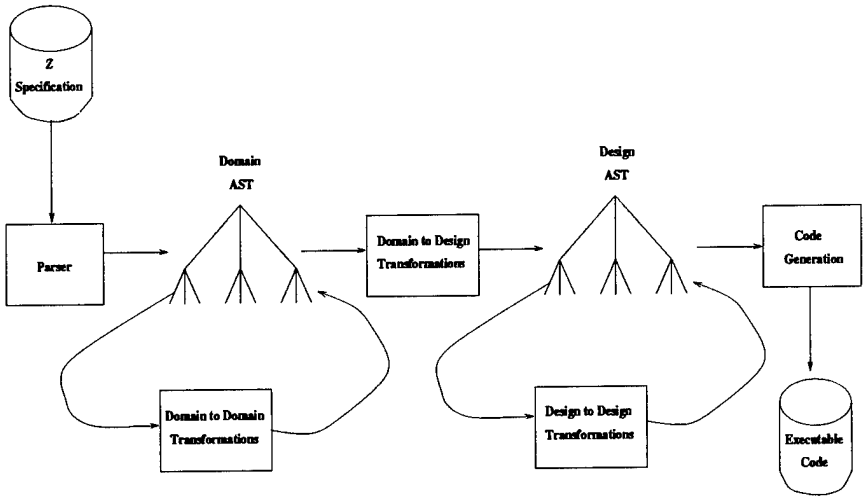


Figure 15 Transformation Process

Since the domain model was in place and functional, one approach to producing a design model would have been to expand this domain model structure into a structure that could represent the design model as well. The final product would have been one wide-spectrum language covering both analysis and design. While this could serve as a design model, it would not present a clear separation between analysis and design. A different approach would be to produce an independent model representing the design model and to build a series of transformations that would populate the design model based on the information in the domain model. This approach would produce a more flexible design

model with one narrow spectrum language for analysis and one for design. By providing a separate model for design there would be a clean break between the thought processes associated with analysis versus design. The Generic Object Model (GOM), designed by Sward [14], is one implementation of a design model, and the one chosen as a building block for this research effort. The original implementation of this model is described here and the final implementation for this research effort is detailed in Chapter IV.

The imperative code structure that was part of the design model needed to be as generic as possible so that multiple target languages could be produced from the design. The methodology used to accomplish this task was one of reviewing the GOM developed by Sward and comparing its composition to the language model summary structure in the Refine manuals representing surface syntax and structure for both Ada [9] and C [10]. The goal was to produce a model that could be considered minimal in terms of the constructs provided. Therefore, the resulting model is less extensive than either of the two complete models, however it is generic enough to allow for implementation in either language. As a result of having a prototype design model in place, the bulk of the analysis in this area focused on the completeness and correctness of the existing model. With this increased emphasis came a number of modifications for polishing the existing GOM into a design model capable of representing many object-oriented languages.

The intent was not to develop an elaborate model for design, but rather a simple structure capable of serving as a minimal model representing many languages. This leaves the designer the flexibility to add those implementation-specific constructs during the transformation from the design model to code. For instance, there are numerous imperative programming constructs that could have been represented in the GOM such as case statements, if-then-else, for loops, etc. Given that all languages do not support all of the constructs available, it was decided that the constructs necessary to represent any input specification should be a minimal set present in every programming language. The programming constructs chosen were assignment, if-then-else, while-loop, procedure-call and return-statement. These basic building blocks are sufficient for the purpose of the design model. The GOM had an output grammar for viewing a populated model in a more

convenient format than by analyzing the AST. This grammar was modified considerably to produce the Ada output.

The first step in understanding this generic design model is to analyze it with relation to how each of its components fit into the object-oriented paradigm. In order to understand the GOM, Sward's dissertation [14] was relied on heavily. The approach taken for presenting this material is to discuss each portion of the object-oriented methodology and show its representative implementation in the GOM. The intent of the GOM is to provide a canonical form for representing a variety of object-oriented languages. As such, each of the following constructs are present in the GOM:

1. Classes - An object class or class describes groups of objects that are similar in properties, semantics and behavior.
2. Objects - Given the above classes, an object is an instance of the class. They break the problem down for easier understanding and implementation.
3. Methods - An operation or a method is a "function or transformation that may be applied to or by objects in a class [3]."
4. Inheritance - All methods and attributes of a parent class are inherited by a child class. The methods can then be specialized and added to meet the needs of the child class.
5. Polymorphism - In the object-oriented paradigm, a different method with the same name can apply to multiple classes. Using polymorphism, a method call will evaluate the parameters of the call to determine the target class of the call and invoke the appropriate method defined by that class definition.

Like the domain model, the GOM is implemented in Refine using Abstract Syntax Tree (AST) structures. Figure 16 defines the grouping of constructs based on inheritance. For example, the GOM-functional-method inherits and specializes the GOM-method. These constructs are defined in detail later in this review. Each construct is approached independently and then shown in full context of its relationship with other entities.

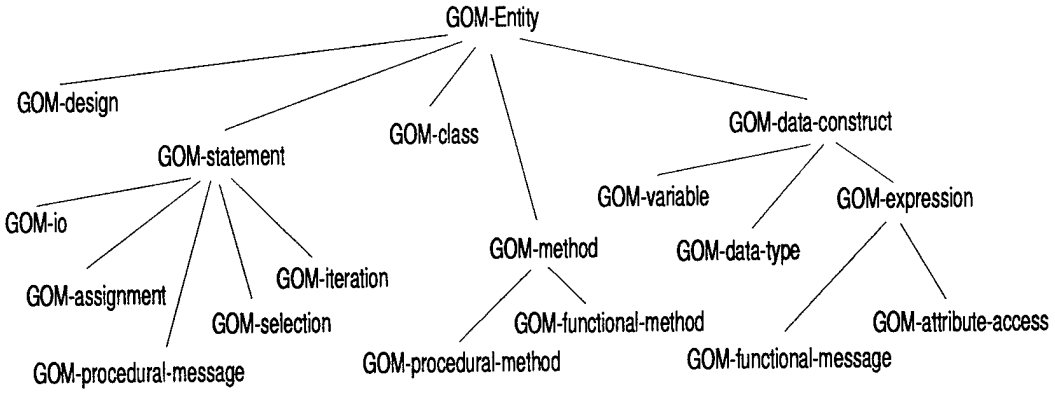


Figure 16 Inheritance Hierarchy for the GOM [14]

3.3.1 Classes. A class defines the template for objects in the system. Classes are represented by the structure of the GOM-Class as shown in Figure 17.

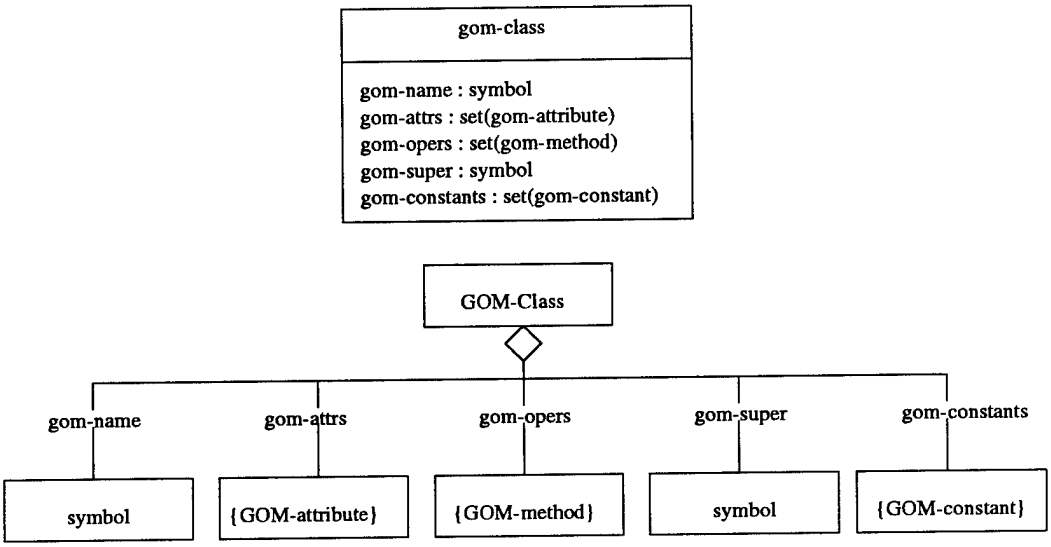


Figure 17 GOM-Class

This construct allows for name, attributes, and inheritance to be directly mapped from the domain model. However, each GOM-Method is constructed from a GOMT-Op in the domain model. States and transitions, which are part of the class structure in the domain model, are notably absent in this construct. This is acceptable due to the fact that they should be transformed into GOMT-Op(s) before being mapped to this model.

3.3.2 Objects. The basic building block for the object-oriented paradigm is the object. "Objects are modeled in the GOM by using variables whose data type refers to a *class*" [14]. An example of an object instance as it would exist in the GOM is shown in Figure 18.

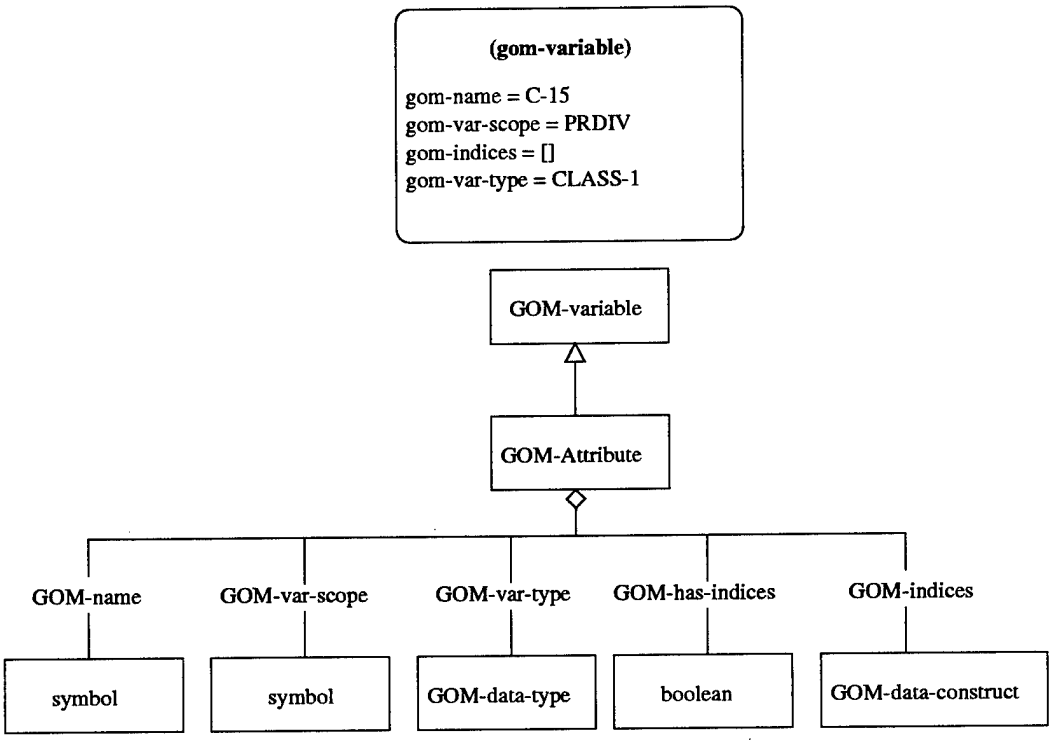


Figure 18 An Instance of a Variable Object [14]

Figure 18 is a decomposition of the GOM-Attribute shown as a component of a class in Figure 17. A GOM-attribute inherits its structure from the GOM-variable allowing instances of attributes to be part of the class structure. The final implementation of the design model, illustrated in Chapter IV, revises this concept of variable.

3.3.3 Methods. A named group of statements that perform a specific function with relation to the class to which they belong is known as a method. There are two different types of methods modeled by the GOM, the procedural method and the functional method. These methods are composed of statements, or GOM-program constructs, and the statements that are "modeled in the GOM include assignment, sequential control flow, selective control flow, iterative control flow and subprogram invocation of non-user-defined

subprograms [14]." Additional program constructs were used in the design model and are covered in Chapter IV. These constructs are implemented in the GOM using the GOM-method construct shown in Figure 19.

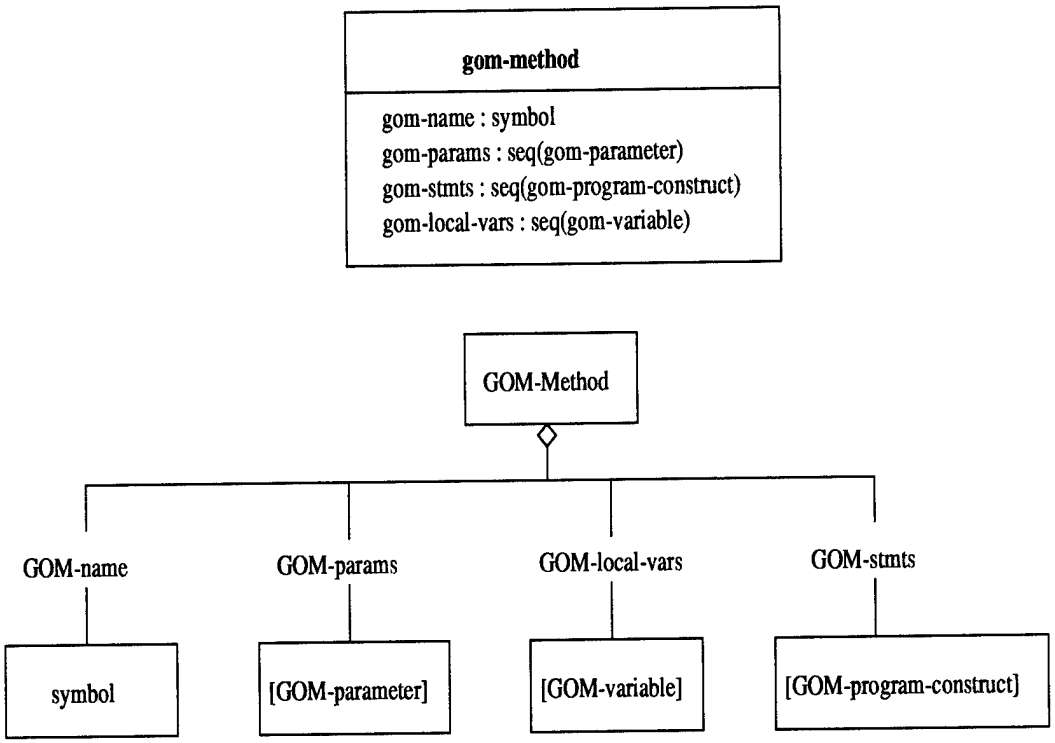


Figure 19 GOM Method

This structure is a decomposition of Figure 17 with the gom-opers of a class being defined as a set of GOM-methods. The structure of the GOM-method relates to the structure of the GOMT-Op with some obvious differences. The set of predicates that compose a GOMT-Op must become a sequence of statements to be executed when the method is called. These statements are represented in GOM-program-constructs. The GOM-method structure does not implement the idea of a method being composed of other method(s) for the purpose of functional decomposition; however, calls to other procedural or functional methods are allowed. Figure 19 represents the "generic" GOM-method; the functional method inherits from it and specializes the structure by adding a return type to the method.

3.3.4 Inheritance. The GOM implements inheritance in the AST by the use of the *gom-super* attribute. This attribute is part of the class structure and can be seen in Figure 17. The gom-super attribute contains a symbol, indicating single inheritance, referencing the class being inherited from.

3.3.5 Polymorphism. The need for declaring methods with the same name for different classes is often necessary in the object-oriented paradigm. For example, most classes will have a method for instantiating a new instance of the class. By using the same name for this method, uniformity throughout the system is gained while providing a unique method for executing this process specific to a class. Polymorphism also provides a mechanism for specializing a method inherited from a super-class. When a method in the sub-class has the same name as a method in the super-class, the sub-class instantiation will invoke the specialized method, providing a service that is unique to the sub-class.

3.3.6 GOM Design. The design model is rooted at the GOM-design object as shown in Figure 20. This provides the basis for connecting the various parts of the object-oriented structure to model a complete system in the GOM AST.

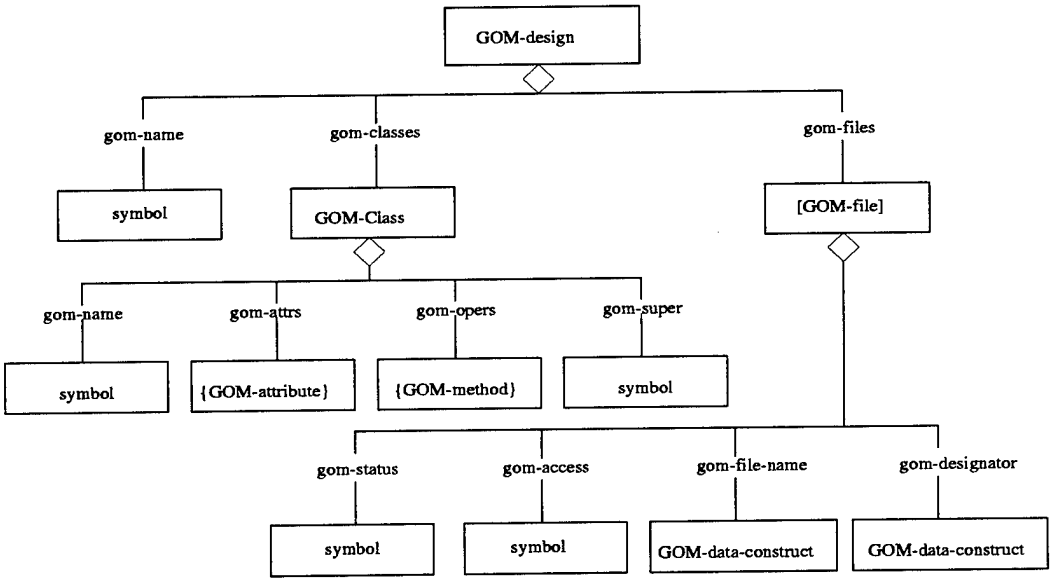


Figure 20 GOM-Design AST

3.4 Transforming the Domain Model to the Design Model

The transforms for converting the domain model into the design model were accomplished in phases. The first phase was concerned with portions of the domain model that are part of the object model. This populated the design model with a minimal set of objects that were then used to produce a skeleton of Ada code. This provided a means of testing and validating the first series of transformations from specification to code. The second phase concentrated on the portions of the domain model that were part of the functional model. When this portion was complete, the design model was populated with a system capable of producing executable Ada source code. The third portion, the dynamic model, was not directly translated from the domain model to the design model. Instead the dynamic model should be used to build an equivalent set of operations in the functional model that can be transformed to the design. This step is left to future research.

Transforming components from the domain model to the design model was approached by building a large number of very specific transformations. Each transformation has a single task of converting an object in the domain model to a specific object in the design model. With these transformations in place, a driver transformation can be called with the populated domain tree and an empty design tree as parameters. It then starts at the root of the domain AST and works its way down through the tree using the specific transformations whenever needed to produce a design model. Each of these transformations including the driver is discussed in detail in Chapter IV.

The predicates were transformed from the domain model to the design model on a one-for-one basis. Each predicate in the domain model became an equivalent construct in the design model. No effort was made to analyze a sequence of statements for optimization or to perform sophisticated algorithm design, which is arbitrarily hard.

3.5 Transforming the Design Model to Source Code

While the goal of this research was to produce compilable and executable Ada source code as a final product, the goal of the tool is to build a generic design capable of producing source code for multiple languages. For the terms of this discussion, Ada was the target

language. There are several ways available to produce Ada code from the design model. Three different approaches were considered in an effort to choose the appropriate one for this research. The first method considered was to write another series of transformations that would walk the design AST and produce the appropriate Ada code. This effort, while feasible, would be considered an extensive task and require major modifications to produce a different language. The second method considered was to use the Refine/Ada AST and to write a series of transformations populating the Ada AST from the design model. The problem with this approach is that the Ada AST only supports Ada 83 and extensions would be needed to support Ada 95. The Ada AST model is extremely complex and the desire of this research was to build a minimal set of constructs. Once this was realized, all of the benefits of actually using the Ada AST were diminished. Finally, the last method considered for producing Ada from the design model was to write an output only grammar. Using an output grammar, the code produced from the design model could be tailored to look like Ada source code, however the formatting of the source code is generally unpredictable and as a result is typically hard to read. Still, the production and maintenance of an output grammar is small in comparison to the two other methods considered, and several example output grammars were available for reference. Based upon these findings, the method chosen to produce Ada from the design model was the output grammar.

As mentioned earlier, some work had previously been accomplished to produce Ada from Sward's GOM [12]. Given the number of changes that were made to the GOM, this work was used only as a reference for the structure of an output only-grammar. The source code produced from the design model targets Ada 95. This provides a functioning system that can take primitive objects from specifications to code with minimal assistance from the designer. The Ada code produced was the final product of this thesis effort.

3.6 Methodology Wrap-Up

In this chapter, a methodology was defined for preparing the domain model for transformation to the design model. Next, a design model had to be defined. This research led to an available model, known as the GOM, which could be modified to fulfill this role.

A three-phase approach was then defined for the transformation of the domain model to the design model. With the design model populated, a method for producing source code was needed. The method chosen was to use an output grammar based on its inherent flexibility.

IV. Implementation

4.1 Overview

The implementation of this research followed the approach defined in Chapter III. The first step in this path was to complete the transformations needed to prepare the domain model for transformation to the design model. This included the addition of the concepts of inheritance to the existing transforms as well as the addition of one new transform. The next step was to validate a design model that could be used in this transformational system. This included some modifications to the GOM [14] which was chosen as the basis for the design model. Next, a set of transformations was created to convert the existing domain model into the design model. While the original goal was to produce a complete design model from this transformation, it was beneficial to copy the predicate structures from the domain to the design. Removing this logic from the domain to design transformation provided the benefit of two smaller, more-manageable sets of transformations. The second set of transformations acts on the design model and converts the predicates copied from the domain model into programming constructs in the design model. Finally, two output grammars were created to produce an Ada 95 specification and an Ada 95 body from the design model.

4.2 Implementation Assumptions

During the planning and implementation phase of this effort, it was necessary to determine a valid set of assumptions or rules that would be enforced. The first such assumption deals with accessing the value associated with an attribute. It was decided that no attribute would be read or updated without the use of the *Set_* or *Get_Attribute* method. This enabled the domain transformations that imposed invariant constraints to concern themselves with only the *Set_* and *Get_Attribute* methods. Any method needing to read or update the value of an attribute would then do so using the appropriate call ensuring the constraints involving that attribute were addressed. The next assumption deals with the correctness of the formal specification parsed into the domain. It was determined that this research would not attempt to validate the correctness of the specification as given. It

is therefore up to the domain expert to provide valid specifications. If invalid specifications are parsed into the system, no statement can be made about the resulting source code.

4.3 Extending Transformations to Handle Inheritance

Inheritance was not considered during the implementation of early transformations, and considerable effort was required to incorporate the idea of inheritance into the transforms. Each transformation was analyzed for its original intent given a system without inheritance and then modified based upon that intent to incorporate the concept of inheritance.

The first transformation modified in the domain model is transform 3. The purpose of this transformation is to handle equality invariant constraints dealing with derived attributes found in the domain model. An invariant constraint is placed upon a class when a condition needs to be true at all times for all members of the class. There are often several ways of ensuring that an invariant constraint is implemented such that it will always be true. The purpose of transform 3 is to enforce that each time one of the attributes from which the derived attribute is derived is changed, the derived attribute is updated and the invariant verified. This involves adding the invariant constraint as a post condition to each *Set_Attribute* method of the attributes involved in the constraint and removing the *Set_Attribute* method for the derived attribute. While this approach is easily implemented in a class without inheritance, consider an inherited attribute. If the derived attribute is inherited, then the *Set_Attribute* method for the derived atttribute does not exist at the sub-class level. In this case the implementation chosen was to build a *Set_Attribute* method for the derived attribute in the sub-class and to make this method a null method. This provides a means for sheltering the sub-class from changing the derived attribute by using a call to the *Set_Attribute* method. Next, consider when one of the attributes that the computed value of the derived attribute is based on is inherited. While no *Set_Attribute* method for this inherited attribute exists at the sub-class level, it would be incorrect to add the invariant constraint to the *Set_Attribute* method at the super-class level where it may have no knowledge of the derived attribute or other attributes involved in the constraint. Therefore, in this situation the implementation chosen was to make a

copy of the *Set_Attribute* method from the super-class to the sub-class and then to add the invariant constraint to the method in the sub-class. This provides a polymorphic approach that specializes the method at the sub-class level.

The next transformation modified in the domain model is transform 4. This transformation is similar to transform 3 in that it deals with equality invariant constraints; however, its approach to this constraint is much different. The purpose of this transform is to recalculate the value of the derived attribute each time it is needed. The attribute and the *Set_Attribute* method are removed from the class since it is no longer needed and the *Get_Attribute* method is modified to compute and return the value of the derived attribute. Again, this approach is not difficult to implement when no inheritance is involved in the constraint, but becomes a bit more tricky with inheritance. If the derived attribute is inherited, then neither the *Get_Attribute* nor the *Set_Attribute* methods for the derived attribute exist at the sub-class level. It is therefore not valid to remove the derived attribute from the class when that attribute is inherited. It is also not valid to remove the *Set_Attribute* method since it resides at the super-class. Therefore, when this situation occurs, an error message is generated to prompt the user indicating transform 3 should be used instead. If the derived attribute is local, but the attributes used to calculate the derived attribute are inherited or consist of a mix of local and inherited attributes, then this transformation executes in the same manner as if no inheritance was involved. The attribute and *Set_Attribute* method are removed from the local class and the *Get_Attribute* method is modified to compute and return the value of the derived attribute.

The next transformation modified in the domain model is transform 5. This transformation also deals with equality invariant constraints, but in a much different way. The purpose of this transformation is to change the value of all attributes associated with the invariant constraint at the same time. This is especially useful when all attributes in the constraint are dependent on the value of each other, for instance if each value represented one angle measurement of a triangle, then the sum of the three attributes must equal 180 degrees. Using this transform, no single attribute could be changed without new values being provided for all angles of the triangle. This is accomplished by building a new *Set_All_Attributes* method containing all of the parameters from each of the individ-

ual *Set_Attribute* methods, all of the predicates from each of the individual *Set_Attribute* methods, and the invariant constraint as a post-condition of the new method. Then the *Set_Attribute* method for each attribute in the constraint is removed from the current class and the new operation is added to the current class. This method increases in complexity with the addition of inheritance. When an attribute involved in the invariant constraint is inherited, its *Set_Attribute* method is copied to the sub-class level. Next, all of the parameters and predicates from this method are included in the new *Set_All_Attributes* method. Then the predicates of the *Set_Attribute* method are removed providing a polymorphic null method at the sub-class level. This implementation handles the problem of inheritance when found in this transformation.

The last pre-existing transformation modified in the domain model is transform 6. This transformation also deals with equality invariant constraints. The purpose of this transformation is to change the value of some of the attributes associated with the invariant constraint at the same time. Restrictions on this transformation include 1) all attributes involved in the invariant constraint must be addressed by this transform and 2) a minimum of two attributes involved in the invariant constraint must be handled together. By approaching an equality invariant constraint in this way, at least two values in the constraint must be changed at once. This is accomplished in a similar manner to transform 5 in that a new *Set_Some_Attributes* method is created that copies the parameters and predicates from each individual *Set_Attribute* method chosen to participate in this new method, the invariant constraint is then added as a post-condition of the new method and the individual *Set_Attribute* methods are removed from the class. This logic is contained in a loop that cycles until every attribute involved in the constraint has been addressed once. This implementation works like transform 5 when inheritance is involved in that when an attribute is inherited, the *Set_Attribute* method for that attribute must be copied to the sub-class where its parameters and predicates can be copied to the applicable *Set_Some_Attributes* method. Then the predicates of the *Set_Attribute* method are removed providing a polymorphic null method at the sub-class level. This implementation handles the problem of inheritance when found in this transformation.

Finally, one new transform, transform 7, was added for the purpose of handling non-equality invariant constraints. The purpose of this transformation was to add the constraint as a post condition to all *Set_Attribute* methods involved in the constraint. This method works by first finding all the *Set_Attribute* methods for those attributes which are not inherited and adding the constraint to them as a post-condition. Next it copies the *Set_Attribute* method of any inherited attribute to the sub-class and adds the constraint as a post-condition of that method. This implementation provides a polymorphic method on inherited attributes.

For each of the transformations discussed, when an invariant constraint is processed by the transformation, it is then removed as an invariant constraint on the class. The manner by which each of the transformations is implemented ensures that the bounds required by the invariant constraint will be properly handled by the methods that modify the attributes upon which the constraint was based. The restriction here is that each attribute be updated or retrieved using the *Set_* and *Get_Attribute* methods for the attribute.

4.4 Design Model

With the population of the domain model in place and functional, the next step of this research was to produce a design model to be used for the transformation from the domain model. The basis for this model is directly attributed to Sward; however, some modifications to this model have been incorporated. This section describes the implementation of the modified model. A description of the original model is found in Chapter II. A complete description is found in Sward's dissertation [14].

When attempting to understand the design of a model whose representation uses an Abstract Syntax Tree (AST), it is necessary to look at the design from two perspectives. The first perspective is the inheritance hierarchy. This shows the hierarchy of object types represented in the model and how like types are grouped together. Given this model, one can see what types inherit from and specialize other types in the model. An illustration of the inheritance hierarchy is shown in Figure 21.

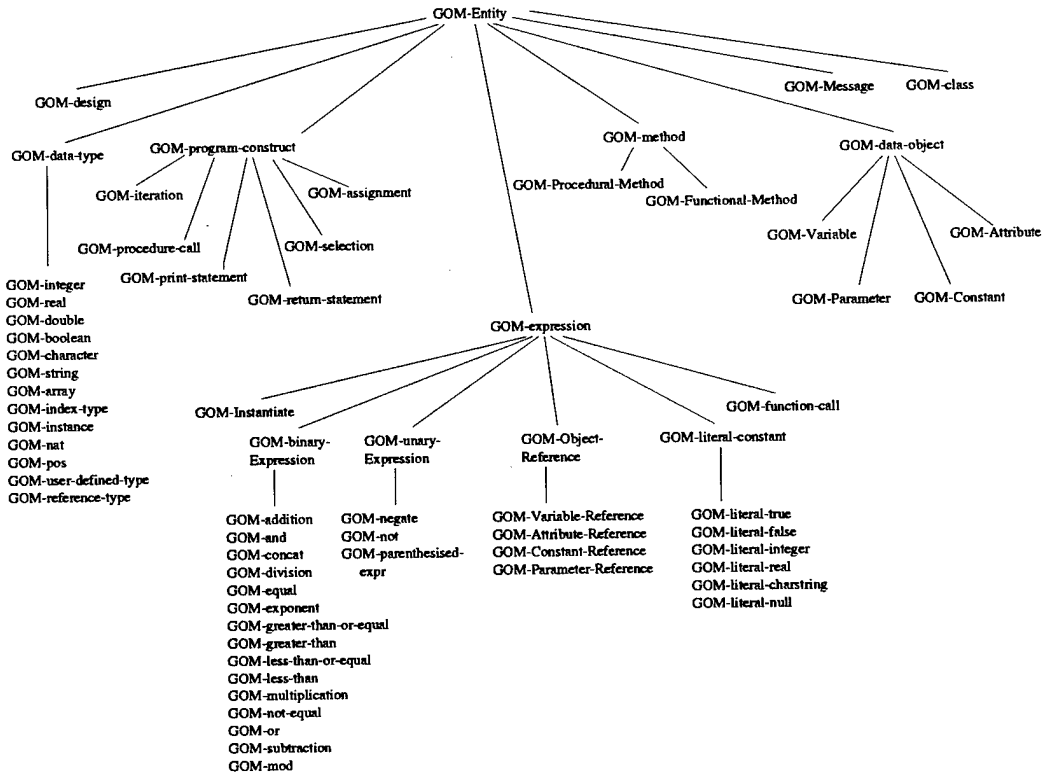


Figure 21 Inheritance Hierarchy for Design Model

Reviewing Figure 21, one can determine that a GOM-addition is a type of GOM-binary-expression which is a type of GOM-expression which is a type of GOM-entity. Note also that one of the available gom-program-constructs is the gom-print-statement. This is a method for producing output from the design model; however, no method exists that accepts input. The reason for this is simply that a formal specification should not be able to specify input or output to the user, but at certain times during the transformations it is helpful to insert informational or error messages to the user. Therefore, the final design model contains output statements but not input statements.

The second perspective is the structural model of the design model. The root of a design is found at the GOM-design object. There is only one GOM-design object per design and all other objects in this model are connected either directly to the GOM-design or through other objects that ultimately are connected to the GOM-design. The structural model for the GOM-design, GOM-class, GOM-attribute and GOM-constant is shown in Figure 22.

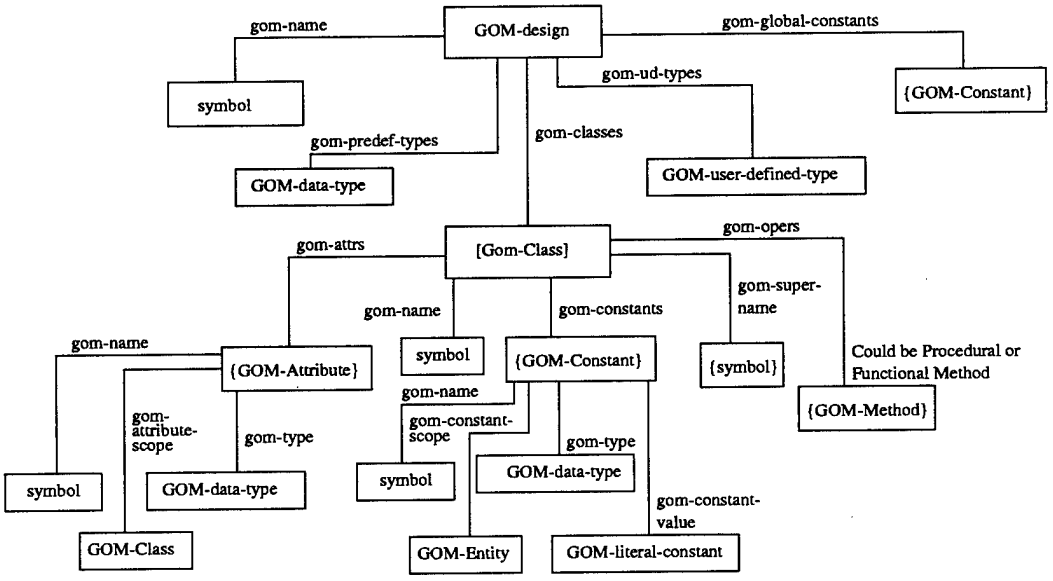
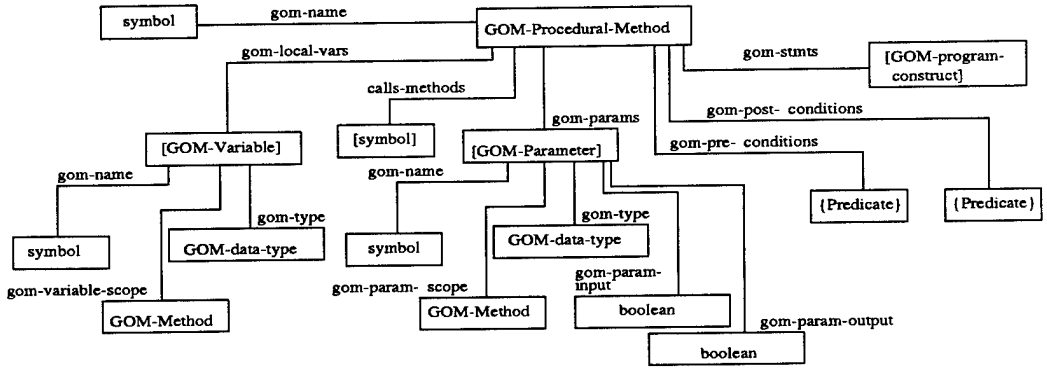


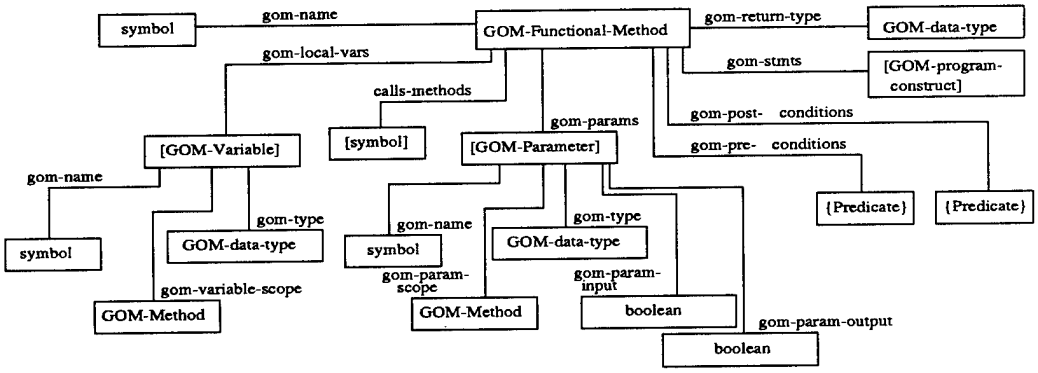
Figure 22 Structural Hierarchy for Design Model

From a review of Figure 22 notice that a GOM-design has a name, global constants, pre-defined and user-defined types, and one or more classes that comprise its structure. A GOM-class is composed of attributes, local constants, methods, and a list of symbols

referencing other classes from which it inherits. While multiple inheritance is supported in this structure, the domain model transformations defined in this research limit inheritance to one level. The names on the lines between objects represent tree attributes that form the links between objects in the AST. Names surrounded by brackets indicate sequences of objects whereas braces indicate sets of objects. Notice that the gom-opers tree attribute at the GOM-class level points to a set of GOM-Method(s). Both a GOM-Procedural-Method and a GOM-Functional-Method inherit from the structure of the GOM-Method and therefore either type of method is valid to insert into this set. The structure of these two types is represented in Figure 23.



a) Structural Model for GOM-Procedural-Method.



b) Structural Model for GOM-Functional-Method.

Figure 23 Structural Model for GOM-methods

A quick review of Figure 23 reveals that the structure of the procedural method and the functional method are almost identical, with the only difference being the gom-return-type tree attribute that adds a pointer to a GOM-data-type object for the functional

method. The GOM-data-type will be found in either the gom-predef-types or gom-ud-types of the design. Otherwise both methods are composed of local variables, parameters, pre- and post-conditions, statements and a sequence of symbols representing the names of other methods called by this method, if any. The purpose of this last attribute is to enable the implementation of a functional hierarchy of called methods. Notice that both gom-pre-conditions and gom-post-conditions connect to a set of predicates. While the predicate is not a construct associated directly with the design model, it does relate to the gom-stmts of each method. Each post condition of a method will become an integral part of the program constructs for that method. This will be explained in detail in section 4.6.2. For now, the makeup of the various legal program constructs of the design model will be covered. Notice in Figure 23 that gom-stmts connects to a sequence of GOM-program-construct. The GOM-program-construct is an abstract class from which all program constructs inherit. A GOM-program-construct can be a GOM-iteration, GOM-procedure-call, GOM-print-statement, GOM-return-statement, GOM-selection, or a GOM-assignment. Figure 24 shows the structure of each of these constructs.

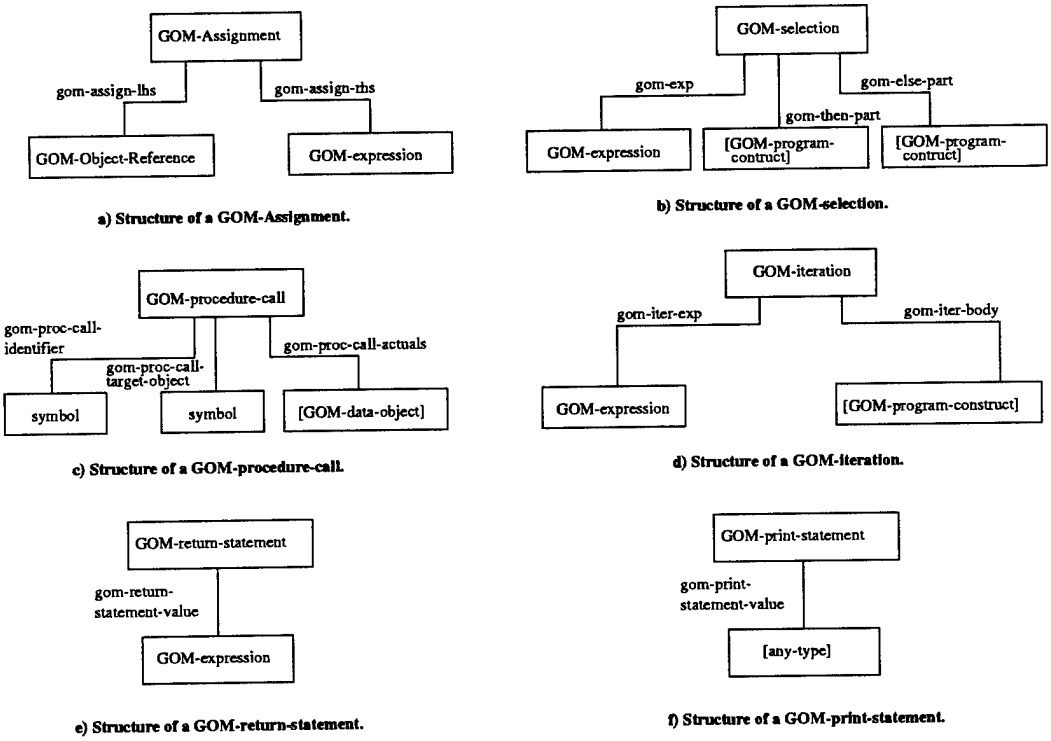


Figure 24 Structural Model for GOM-program-constructs

Reviewing Figure 24a, notice that the left hand side of a GOM-Assignment can be any object reference. This allows this value to be an attribute, variable, parameter or constant reference. Of course, it would only be valid for a constant during initialization. Also note that some program constructs are themselves composed of program constructs allowing for nesting. Because of this, the design model can get by supporting only the if-then-else construct without need for else-if constructs. With a general knowledge of the domain model, the transformations executed on the domain model and structural knowledge of the design model, next consider transforming the domain model into a design model.

4.5 Transforming the Domain Model to the Design Model

The intent of this task was to build a large number of very specific transformations that could be called recursively to produce a design structure from a domain structure. The driver calls each transformation that directly affected its components and likewise each transformation called calls lower-level transformations for each of its components. The design transformation was the driver for this series of transformations, but also necessary to accomplish this task were more detailed transformations for classes, global and local constants, pre-defined and user-defined types, methods, parameters, predicates, attributes and super-classes. Each one of these lower level transforms is defined in detail in the following sections.

4.5.1 Design Transformation. This transformation is the driver for this series of transforms. When called, this transformation is passed as parameters the populated domain AST and an empty design AST. Based on the information found in the domain, a design is constructed. The steps of the design transformation are as follows:

1. Assign the design tree the same name as the domain tree.
2. For every pre-defined type found in the domain, call the pre-defined type transformation and place the resulting pre-defined type design structure into the gom-predef-types attribute of the gom-design. Since the domain had no concept of a string type as a pre-defined type, add this type to the gom-predef-types also.

3. For every user-defined type found in the domain, call the user-defined type transformation and place the resulting user-defined type design structure into the gom-ud-types attribute of the gom-design.

4. For every primitive-class found in the domain, call the primitive class transformation and place the resulting gom-class into the gom-classes attribute of the design. For a description of the transformations for aggregate classes, see Kissack [4].

5. Once all classes have been converted from the domain to the design, the links between classes involved in an inheritance relationship are established to reflect the inheritance found in the domain.

6. For every constant existing at the global level of the domain, call the global constant transformation and place the resulting gom-constant into the gom-global-constants attribute of the gom-design.

7. For every symbol found in the design model, convert each lower-case letter to upper-case and covert all hyphens to underscores.

Step seven was implemented to produce a prettier output when using the grammar. Also, the hyphen found in many of the symbols is not parsable by some target languages.

4.5.2 Pre-defined Type Transformation. When called, this transformation is passed a pre-defined type from the domain. The steps of the pre-defined type transformation are as follows:

1. Instantiate an empty gom-data-type.

2. Test the name of the pre-defined type against the known names of pre-defined types existing in the domain. When a match is found, instantiate the corresponding type from the design and replace the type from step 1 with a new instantiation.

3. Return the new gom-data-type to the caller transform.

4.5.3 User-defined Type Transformation. When called, this transformation is passed a user-defined type from the domain. The steps of the user-defined type transformation are as follows:

1. Instantiate an empty gom-user-defined-type.
2. Assign the gom-user-defined-type the name from the domain user-defined-type.
3. If the user-defined-type from the domain is of DomDerType, meaning it was derived from another type, assign the gom-derived-from attribute of the gom-user-defined-type to be equal to the has-datatype attribute of the domain's user-defined-type.
4. If the domain's user-defined-type has any enumeration values, copy these values into the gom-enumeration-values of the gom-user-defined-type.
5. If the domain's user-defined-type is constrained by any predicates, copy these predicates into the gom-type-constraints of the gom-user-defined-type.
6. Return the gom-user-defined-type to the caller transform.

4.5.4 Primitive Class Transformation. When called, this transformation is passed a primitive class from the domain. The steps of the primitive class transformation are as follows:

1. Instantiate an empty gom-class and assign it the name of the class from the domain.
2. For every attribute found in the domain's class, call the attribute transformation.
3. For every local constant found in the domain's class, call the private constant transformation.
4. For every operation found in the domain's class, call the operation transformation.
5. Return the new gom-class to the caller transform.

4.5.5 Attribute Transformation. When called, this transformation is passed an attribute from the domain. The steps of the attribute transformation are as follows:

1. Instantiate an empty gom-attribute and assign it the name of the attribute from the domain.
2. Locate the pre-defined or user-defined type that applies to this attribute and place a pointer from the attribute to this type in the gom-type attribute of the new gom-attribute.

3. Set the scope of the gom-attribute equal to the gom-class in which it resides.

4. Insert the new gom-attribute into the gom-attrs attribute of the gom-class.

4.5.6 Private Constant Transformation. When called, this transformation is passed a constant from the domain. This constant is local to a class. The steps of the private constant transformation are as follows:

1. Instantiate an empty gom-constant and call the build-constant function that is shared by the global and private constant transformations.
2. Set the scope of the gom-constant equal to the gom-class in which it resides.
3. Insert the new gom-constant into the gom-constants attribute of the gom-class.

Note that the build-constant function determines the type for the gom-constant and builds a link between it and the appropriate type definition. It then sets the value of the gom-constant equal to the value of the domain's constant and returns the gom-constant to the caller transformation.

4.5.7 Operation Transformation. When called, this transformation is passed an operation from the domain. The steps of the operation transformation are as follows:

1. Instantiate an empty gom-procedural-method and assign it the name of the domain's operation.
2. For every parameter found in the operation from the domain, call the parameter transformation.
3. If this operation calls other operations, copy this information to the calls-methods attribute of the gom-method.
4. Add the new gom-procedural-method to the gom-opers attribute of the gom-class.
5. For every predicate found in the domain's operation, call the predicate transformation.

4.5.8 Parameter Transformation. When called, this transformation is passed a parameter from the domain. The steps of the parameter transformation are as follows:

1. Instantiate an empty gom-parameter and assign it the name of the domain's parameter.
2. Set the scope of the parameter equal to the gom-procedural-method.
3. If the is-output attribute of the domain parameter is false or undefined, then set the input flag of the gom-parameter to true and the output flag to false. Otherwise, set the input flag of the gom-parameter to false and the output flag to true.
4. Search the pre-defined and user-defined types of the design model for the type that applies to this parameter and establish a link from the gom-parameter to the appropriate type definition.
5. Add the new gom-parameter to the gom-params attribute of the gom-procedural-method.

4.5.9 Predicate Transformation. When called, this transformation is passed a predicate from the domain. The steps of the predicate transformation are as follows:

1. Make a copy of the domain's predicate.
2. If the predicate is decorated with final decorations indicating that some value in the predicate will be changed, it is considered a post-condition. Also, if any output parameters of the method appear in the predicate, it is considered a post-condition. Otherwise, it is considered a pre-condition.
3. Pre-conditions are placed in the gom-pre-conditions attribute of the gom-procedural-method. Post-conditions are placed in the gom-post-conditions attribute of the gom-procedural-method.

4.5.10 Global Constant Transformation. When called, this transformation is passed a constant from the domain. The steps of the global constant transformation are as follows:

1. Instantiate an empty gom-constant and call the build-constant function.
2. Set the scope of the constant equal to the gom-design.
3. Add the gom-constant to the gom-global-constants attribute of the gom-design.

The structure making up the domain model was transformed into a viable design structure using the transformations listed above. This is not the end of the transformations needed to produce compilable code. The next series of transformations discussed concerns transforming constructs found in the design model to other design constructs necessary to produce compilable code.

4.6 Transformations Applied to the Design Model

In section 4.2, the assumptions that guided this research effort were addressed. One of these determined that no attribute should be accessed without the use of the *Set_* or *Get_Attribute* method for that attribute. This assumption led to the need to modify any attribute reference found in a program construct, when that construct was not a part of the *Set_* or *Get_Attribute* method for that attribute, to make calls to those *Set_* or *Get_Attribute* methods. Also, all operations found in the domain model are initially transformed into procedural methods in the design model. A transformation was implemented to sort out the functional methods from the procedural methods. Finally, the post-conditions that were copied from the domain model to the design model are transformed into executable statements. This transformation process will transform each post-condition into a single statement. These problems and others are the focus of this section.

4.6.1 Transforming Procedural Methods to Functional Methods. The first transformation designed and implemented on the design model had the purpose of converting all *Get_Attribute* methods into functional methods of the design. As each *Get_Attribute* method only returns one value, the stored or calculated value of an attribute, it naturally fit the mold of a functional method and should be transformed. This transformation is only applied to the *Get_Attribute* methods currently; however, any method with only one output parameter could feasibly be transformed using this transformation. The steps of this transformation are as follows:

1. For each *Get_Attribute* procedural method in the design, instantiate an empty gom-functional-method and assign it the name of the *Get_Attribute* procedural method.
2. Add an instance parameter of the method's class as the first parameter of the method.
3. If any local variables existed in the procedural method, copy them to the functional method.
4. Copy the pre- and post-conditions of the procedural method to the functional method.
5. Set the return type of the functional method equal to the datatype of the output parameter of the method.
6. Transform the post-condition of the functional method into a gom-return-statement.
7. Add the gom-return-statement to the gom-stmts of the functional method.
8. Remove the procedural method and add the functional method to the gom-opers attribute of the class it belongs to.

4.6.2 Transforming Post-Conditions into GOM Constructs. The design model is now prepared to transform the predicates copied over from the domain model into GOM constructs in the design model. As the title suggests, only post-conditions are transformed. There is no transform for dealing with the pre-conditions. One of the assumptions made during this research effort was that this system would work under the premise of "contractual" programming. This means that any method will produce proper results as long as the inputs given to the method meet the pre-conditions of the method. If the pre-conditions are not met, then no guarantees are made. Also, during this transformation many of the GOM constructs produced are not GOM-program-constructs shown in Figure 24. For instance, many predicates when transformed produce expressions rather than program constructs. At this stage of transformation, all of these constructs are placed in the gom-stmts attribute of the method regardless of the type of construct it is. During a later transformation, these situations are analyzed and dealt with.

A select group of predicates and expressions have been chosen for this effort. There are many other predicates and countless combinations of those predicates which are not part of this implementation. A system capable of handling all predicates and combinations

of predicates would be doing algorithm design. This task is arbitrarily difficult and beyond the scope of this research effort. What is shown in this section, however, is a method for transforming a select group of predicates and expressions into a respective set of program constructs in the design model. The predicates and expressions addressed were chosen because they frequently occur in primitive classes. Even within this select group of predicates and expressions, all cases cannot be handled by this transformation system. When this occurs, it is the goal of this transformation system to signal the user that the system is in over its head. In primitive classes these situations should be rare and most of the primitive examples produce meaningful, well-structured code. The next transformation to be analyzed deals with the task of transforming the post-conditions of a method into gom-program-constructs. For each procedural method containing post-conditions the following steps occur:

1. Add an instance parameter of the gom-class containing the method as the first parameter of the method.
2. For each post-condition associated with the method, perform a test to determine the type of predicate and call the corresponding transform as shown in Table 1.
3. Place the program construct returned from the transform into the gom-stmts attribute of the procedural method.

Each transform listed in Table 1 is explained in detail in the following sections with examples as necessary. Some of these transformations are similar in scope and purpose and are described together.

4.6.3 True and False Predicate Transformation. When a predicate transformation recognizes that one of its components is either a true or false predicate, it then calls the respective transformation and passes that portion of the predicate to the called transformation. The transformation then instantiates a gom-literal-true or gom-literal-false that is then returned to the caller transform.

4.6.4 Bracket Predicate Transformation. When a predicate transformation recognizes that one of its components is a bracket predicate, it then calls the bracket predicate

Table 1 Mappings between Predicates, Transforms and Design Constructs

Predicate found	Transform called	GOM construct produced
true-pred	do-true-xform	GOM-literal-true
false-pred	do-false-xform	GOM-literal-false
bracket-pred	do-bracket-xform	GOM-parenthesized -expr
negated-pred	do-negated-xform	GOM-not
implication-pred	do-implication-xform	GOM-selection
equivalent-pred	do-equiv-xform	GOM-and [1]
schema-ref-pred	print statement	[2]
disjunct-pred	do-disjunct-xform	GOM-or
conjunct-pred	do-conjunct-xform	GOM-and
relational1-pred	do-relational1-xform	See Table 2.
relational2-pred	print statement	[3]
relational3-pred	print statement	[3]

[a]The equivalent predicate equates to a boolean "if and only if" construct. Given this information, the do-equiv-xform transform transforms A "iff" B into (((not A) or B) and ((not B) or A)) with the gom-and being the root of the expression.

[b]The schema-ref-pred is essentially an attribute, variable, parameter or constant reference and as such can be handled by lower level transforms, but as a root level predicate they supply no logic for transformation.

[c]The relational2- and relational3-pred are similar to the relational1-pred with the exception that they have more relational symbols and expressions. These predicates, while valid, are discouraged and therefore no transformations have been supplied for them.

transformation and passes that portion of the predicate to the called transformation. This transformation then executes the following steps:

1. Instantiate an empty gom-parenthesized-expr.
2. Test the predicate contained inside the bracket-pred for its type and call the appropriate transformation to convert it.
3. Place the results returned from the transformation into the gom-unary-operand attribute of the gom-parenthesized-expr.
4. Return the gom-parenthesized-expr to the caller transformation.

4.6.5 Negated Predicate Transformation. When a predicate transformation recognizes that one of its components is a negated predicate, it then calls the negated predicate transformation and passes that portion of the predicate to the called transformation. This transformation then executes the following steps:

1. Instantiate an empty gom-not expression.
2. Test the predicate contained inside the negated-pred for its type and call the appropriate transformation to convert it.
3. Place the results returned from the transformation into the gom-unary-operand attribute of the gom-not.
4. Return the gom-not expression to the caller transformation.

4.6.6 Implication Predicate Transformation. When a predicate transformation recognizes that one of its components is an implication predicate, it then calls the implication predicate transformation and passes that portion of the predicate to the called transformation. This transformation then executes the following steps:

1. Instantiate an empty gom-selection, if-then-else construct.
2. Test the left side of the implication for its type and call the appropriate transformation to convert it.

3. Place the results returned from this transformation into the gom-exp attribute of the gom-selection.
4. Test the right side of the implication for its type and call the appropriate transformation to convert it.
5. Place the results returned from this transformation into the gom-then-part attribute of the gom-selection.
6. Return the gom-selection statement to the caller transformation.

For instance, this transformation would transform an implication predicate of the form $(a \Rightarrow b)$ would result in a gom-selection of the form "if a then b".

4.6.7 Equivalent Predicate Transformation. When a predicate transformation recognizes that one of its components is an equivalent predicate, it then calls the equivalent predicate transformation and passes that portion of the predicate to the called transformation. This transformation then executes the following steps:

1. Instantiate one gom-and, two gom-or(s) and two gom-not(s).
2. Link these five structures together as shown in Figure 25.
3. Test the left side of the equivalent predicate for its type and call the appropriate transformation to convert it.
4. Place the results returned from this transformation into the objects labeled 1 and 4 in Figure 25.
5. Test the right side of the equivalent predicate for its type and call the appropriate transformation to convert it.
6. Place the results returned from this transformation into the objects labeled 2 and 3 in Figure 25.
7. Return the gom-and expression, the root of the tree, to the caller transformation.

For instance, this transformation would transform an equivalent predicate of the form $(a \Leftrightarrow b)$ to "(((not a) or b) and ((not b) or a))" in the design model.

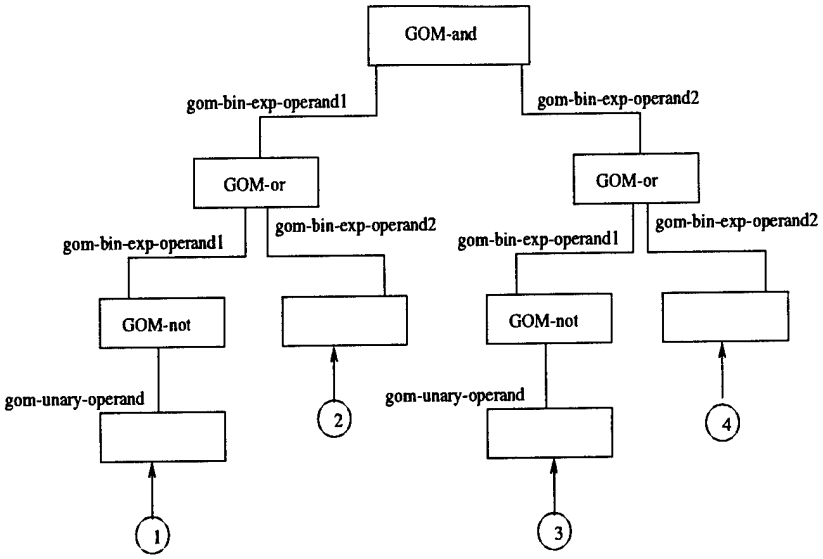


Figure 25 Design Model Structure of a Domain Model Equivalent Predicate

4.6.8 Disjunct and Conjunct Predicate Transformation. When a predicate transformation recognizes that one of its components is either a disjunct or conjunct predicate, it then calls the respective transformation and passes that portion of the predicate to the called transformation. This transformation then executes the following steps:

1. Instantiate either a gom-or for the disjunct or a gom-and for the conjunct predicate.
2. Test the left side of the predicate for its type and call the appropriate transformation to convert it.
3. Place the results returned from this transformation into the gom-bin-exp-operand1 attribute of the gom-and or gom-or.
4. Test the right side of the predicate for its type and call the appropriate transformation to convert it.
5. Place the results returned from this transformation into the gom-bin-exp-operand2 attribute of the gom-and or gom-or.
6. Return the gom-and, or gom-or, expression to the caller transformation.

4.6.9 Relational1 Predicate Transformation. The relational1-pred cannot be directly transformed into a design construct based upon the type of predicate alone. The

relational symbol contained in the predicate determines the resulting design construct. Table 2 shows the relational symbol, the transform used to convert it, and the resulting design construct.

Table 2 Mappings from Relational1 Predicates, Transforms and Design Constructs

Relational Symbol	Transform called	Design construct produced
$\neq$	do-ne-xform	GOM-not-equal
$=$	[1]	[1]
$>$	do-gt-xform	GOM-greater-than
$\geq$	do-gte-xform	GOM-greater-than-or-equal
$<$	do-lt-xform	GOM-less-than
$\leq$	do-lte-xform	GOM-less-than-or-equal

[a]When the relational symbol is found to be "equal" two situations can occur. First, if a final decoration is found in the predicate indicating the change of a value after the execution of the predicate, then it becomes a GOM-assignment program construct and the transform called is do-eq-xform. Second, if no final decoration is found in the predicate indicating that this predicate is simply a boolean test for equality, then it becomes a GOM-equal expression and the transform called is do-equality-xform.

4.6.10 Transformations for Expressions. Most predicates are composed of other predicates and therefore each transform discussed has a recursive nature. The previous section covered all of the predicate transforms in detail; however, the relational1 predicate is composed of expressions. Therefore, these expression transformations and their steps are covered next.

The greater than, greater than or equal, less than, less than or equal and the not equal expression transformations are similar in operation, but return different type of gom-expressions depending on which one is called. The greater than transformation returns a GOM-greater-than expression. The greater than or equal transformation returns a GOM-greater-than-or-equal expression. The less than transformation returns a GOM-less-than expression. The less than or equal transformation returns a GOM-less-than-or-equal expression. The not equal transformation returns a GOM-not-equal expression. The steps below describe all five transformations:

1. Instantiate an empty gom-expression of the appropriate type.

2. Test the left side of the relational1 predicate for its type and call the appropriate transformation to convert it.
3. Place the results returned from this transformation into the gom-bin-exp-operand1 attribute of the gom-expression.
4. Test the right side of the relational1 predicate for its type and call the appropriate transformation to convert it.
5. Place the results returned from this transformation into the gom-bin-exp-operand2 attribute of the gom-expression.
6. Return the gom-expression to the caller transformation.

As predicates were composed of other predicates, expressions are composed of other expressions. This means that these transforms also need to be recursive and test for each type of expression of which they could be composed. The additional expressions that this transformation system is capable of recognizing and transforming are listed in Table 3.

Table 3 Mappings between Expressions, Transforms and Design Constructs

Expression Found	Transform called	GOM construct produced
Addition Expression	do-add-xform	GOM-addition
Subtraction Expression	do-sub-xform	GOM-subtraction
Variable Name Expression	do-var-xform	GOM-object-reference
Modulo Expression	do-mod-xform	GOM-mod
Multiplication Expression	do-mult-xform	GOM-multiplication
Division Expression	do-div-xform	GOM-division
Exponent Expression	do-exp-xform	GOM-exponent
Integer Expression	do-integer-xform	GOM-literal-integer
Real Expression	do-real-xform	GOM-literal-real

The addition, subtraction, modulo, multiplication, division and exponent expression transformations all produce binary expressions using the same technique as the greater than transformation. The type of expression returned by each of these transformations can be found in Table 3.

The variable name transformation's name is misleading. A variable name found in a expression can represent several things. It could be a variable, parameter, constant,

attribute, or maybe an enumerated value. Therefore, considerable logic is contained in this transformation. The steps taken by this transformation are as follows:

1. Instantiate an empty object reference.
2. Scan the design tree to determine if the variable name represents a constant, attribute, parameter, or local-variable in that order. Once its roots are determined, the appropriate reference type is built and given the name of the variable.
3. The object-reference is returned to the caller transform. If the name is not found, only the symbol of the variable name is returned.

The integer and real expression transformations do not have to check for further decomposition. They simply instantiate either an empty gom-literal-integer or gom-literal-real, assign the new literal the value associated with the integer or real expression, and return the gom-literal to the caller transformation.

This concludes the topic of domain construct to design construct transformations. With these transformations finished, the gom-stmts attribute of each method is now populated with design constructs derived from the post-conditions. The initial format of these design constructs does not meet the specifications required by this transformation system. Therefore, the following sections detail the additional transforms needed to complete the process.

4.6.11 Transforming Expressions into GOM-Selection Statements. Continuing to transform each GOM construct into its proper format, each sequence of GOM constructs found in the gom-stmts attribute of a method must be analyzed. Remember that expressions are not valid GOM-program-constructs; however, when transforming the relational1-pred in the previous transform it is possible that some expressions could have been stored in the gom-stmts attribute of various methods. When this occurs the do-expr-xform locates the situation and corrects it. It first creates a gom-selection program construct, which is an if-then-else. It then places expressions found in the gom-stmts attribute of the method into the gom-exp attribute of the gom-selection. If more than one expression is found in the gom-stmts, they are all placed into the gom-exp attribute of the gom-selection and

joined by the boolean "and". Next, it places all other gom-stmts of the method in the gom-then-part of the gom-selection and builds an error statement to be printed when the gom-exp of the gom-selection fails and puts it into the gom-else-part of the gom-selection. All individual constructs are then removed from the gom-stmts of the method and replaced with the new gom-selection.

For example, if two constructs are found in the gom-stmts of a method and the first one is "A' $\leq$ 100" and the second one is "A' = X_A?" where X_A? is an input parameter whose purpose is to change the value of A, then the resulting construct will be "If X_A $\leq$ 100 then A := X_A". Notice that the expression "A' $\leq$ 100" was converted to a pre-condition testing the value of the input parameter before actually changing the value of A.

4.6.12 Transforming the Initialization Method. The initialization method for each class is considered a special case for several reasons. First, each attribute of a class should be initialized using the initialization method; however, the *Set_Attribute* method for each attribute being initialized should not be called. This is because the constraints that may be involved in the *Set_Attribute* method could prevent some attributes from being initialized at all (i.e., one at a time). Therefore, the initialization method should use a direct attribute reference for initializing each attribute. Second, during the transformations that prepared the domain model for conversion to the design model, it is possible that some attributes were completely removed from the class structure. This transformation must validate that each attribute being initialized does indeed exist and if not, it must remove the assignment statement concerning that attribute. Finally, some attributes that need to be initialized for the current class could be inherited attributes and as such would most likely not have initialization values at the sub-class level. In either case, this transformation would insert a call to the initialization method of the super-class as the first statement of the initialization of the sub-class. In this manner, if the sub-class' initialization method does set a value for the inherited attribute, it will overwrite the value assigned by the call to the initialization method of the super-class. The initialization method is the only

method that is not required to use the *Set_Attribute* method to change the value of an attribute when it is available.

4.6.13 Transforming Attribute References into Method Calls. Given the assumption that no attribute should be directly accessed when a *Set_* or *Get_Attribute* method exists for that attribute, this transformation ensures that this rule is complied with. It is actually two separate transformations.

The first transformation scans the design model looking for attribute references found in expressions. When this situation occurs, the transformation builds a function call to the *Get_Attribute* method, validating that this function does exist and whether or not it is a part of the current class or a super class, and then it replaces the attribute reference in the expression with the function call. This transformation ensures that no attribute is directly referenced when a call should be made to the *Get_Attribute* method.

The second transformation will scan the design model looking for assignment statements. As each assignment statement is found, it must be determined that it is not part of the initialization method for the class or the *Set_Attribute* method for the attribute being referenced. When neither of these conditions exist, the transformation then builds a procedure call to the *Set_Attribute* method of the attribute found in the reference. Next, it validates that the *Set_Attribute* method does exist and whether it is local or inherited. It then builds a parameter reference to the instance variable of the class to insert as the first parameter of the procedure call. The second parameter of the procedure call is the right hand side of the assignment statement. Finally, the procedure call replaces the assignment statement in the design model wherever it existed previously. Remember that gom-program-constructs exist as part of other gom-program-constructs as well as stand alone gom-stmts; therefore, the newly generated procedure call must replace the assignment statement at whatever location it previously existed.

4.6.14 Transforming the Null Method. It is possible to have methods existing in the domain model that contain no post-conditions. If a method has no post-conditions, it will not have any program constructs in its gom-stmts attribute either. This could pose a

problem to target language compilers if they do not know how to handle an empty method. However, this transformation will locate these special methods in the domain model and add an instance parameter to the method and a null operation to the gom-stmts attribute of the method. This preserves the method in case it was intended to be a stubbed procedure and also allows for compilation of the method with no functionality.

4.6.15 Transforming the Unknown Type. During the specification of a system, it is possible to define a type that is not derived from or has no links to other pre-defined types in the domain. This is an acceptable approach to domain modeling; however, during design a decision must be made as to the derivation of these types. This transformation will search the design tree for types that do not have enough information to indicate their heritage and will prompt the user for a pre-defined type that this unknown type should be derived from. As a result this unknown type becomes a sub-type of the type selected by the user. Ideally this interface with the user would list both pre- and user-defined types to choose from; however, only pre-defined types were covered by this effort. The interface between the system and the user is shown below:

What would you like to do with type unknown-type?

0 - Return to main menu.

1 - Make Type a Natural Integer(0, 1, 2, ...)

2 - Make Type a Positive Integer.

3 - Make Type an Integer.

4 - Make Type a Character.

5 - Make Type a String.

6 - Make Type a Boolean.

7 - Make Type a Real.

This menu allows the user to pick the best pre-defined type to derive the unknown type from.

4.6.16 Transformations on String Types. In the predicates discussed in the domain model, there was no concept of a string type; however, if an assignment statement contains an attribute reference on the left hand side and the type of the attribute is a string then it needs to be determined if the right hand side of the string must be transformed. First, if the right hand side of the assignment is an input parameter to the method, then no changes need be made. Otherwise, the right hand side will be modified to appear as an Ada unbounded string. The only types of strings allowed in the design concept are unbounded strings and as a result the assignment statement will look like "cls.attribute := to_unbounded_string("string");".

This is one instance where the design model is Ada specific. It was a continuing task to keep the Ada specifics of the model to a minimum. The vast majority of the Ada specifics are maintained in the output grammars used to produce the Ada code for this system.

4.7 Producing Ada from the Design Model

With the transformations complete, the design model is now prepared to produce Ada source code. Several methods exist for accomplishing this task. The use of output grammars was chosen as it is similar to using a context-free grammar, which is a powerful tool for analyzing and describing languages [6]. For this task it was necessary to produce two separate grammars. The first grammar describes the method for producing the Ada body from the given domain model and the second grammar describes the method for producing the Ada specification from the same domain model.

Each class that exists in the domain model produces a class-name.adb Ada body file and a class-name.ads Ada specification file. Also produced is a design-name-udtypes.ads file that declares all of the user defined types used by the design model. Each class specification or body file will "with" the user-defined types specification when needed.

The grammars used to produce these files were written and designed using the Dialect tool built by Software Refinery [7]. This tool allows grammars to be specified for both input to an AST and output from an AST. While the definition of an input grammar, or parser,

is quite complicated, the definition of an output-only grammar, while still complex, is not as bad. For instance to produce the Ada body using Dialect the output grammar begins at the class level of each class in the domain model. It uses the following statements at the class level of the output grammar:

GOM-Class ::= [["with Ada.Text_IO, Ada.Strings.Unbounded;"]

["use Ada.Text_IO, Ada.Strings.Unbounded;"]

["package body" gom-name "is"

{["–superclasses:" gom-super-name * ""]}

{["–local constants" gom-constants * ""]}

["–methods" gom-opers + ""]]

"end" gom-name ";"]

print-only,

The output file specified by the driver program of the print utility begins the Ada file with the "with" and "use" statements listed in the top two lines of the grammar. Next, it inserts "package body", then the gom-name of the class and the keyword "is". For sake of reference, if this class inherits from a super-class, this class is listed as a comment in the output file. Next, it determines if the current class contains any local constants and if it does, it uses the production listed in the grammar for gom-constants to print the declaration of these constants in the output file. The * following the field gom-constants indicates that there may be zero or more local constants associated with this class. If this value were +, this would indicate that there would be at least one, but possibly more. After that, each method is printed in the output file using the production listed in the grammar for gom-opers. Remember that there were two different types of methods in the design model, a procedural method and a functional method. Either method fires off a call to produce an output and while they both are inherited from the abstract object GOM-method, the GOM-Method has no production in the grammar. However, there are separate production for the GOM-procedural-method and the GOM-Functional-Method and based upon the type of method the proper production is used.

4.8 Limitations

This effort focused on transforming constructs from the domain model to the design model that were frequently found in primitive objects. There are, however, many additional predicates and expressions that were not transformed during this effort. Some of these constructs should only be found in aggregate objects and as such would be outside the scope of this research; however, many others could be found in primitive classes but are rarely used. Table 4 lists the predicates found in the domain model that were not transformed to design constructs while Table 5 lists the expressions in the domain model not transformed. While this list appears quite large, it represents all predicates and expressions that can be specified using the *Z* language. The *Z* parser that populates the domain tree will not recognize many of these structures and most of the rest again relate primarily to aggregate objects. For more information concerning predicates and expressions for aggregate objects that can be transformed, see Kissack [4].

Table 4 Predicates Not Transformed

universal-pred
existential-pred
unique-pred
disjointset-pred
preschemaref-pred

Within the bounds of the predicates and expressions transformed during this research, there are other situations that cannot be appropriately handled by this system. For instance, consider a specification for a square root method consisting of one input parameter, x?, one output parameter, y!, and a single post-condition, y! * y! = x?. While this is a valid specification, the existing transformations could not provide a square root function from it.

Next, consider a specification for a swap method containing two predicates. The first predicate would specify x' = y, and the second predicate would specify y' = x. Again, this specification is valid, but the resulting design specification would assign the value of y to

Table 5 Expressions Not Transformed

lambda-expr	mu-expr	relation-expr	z-function-expr
injection-expr	partialinject-expr	surjection-expr	partialsurject-expr
bijection-expr	finitepartfun-expr	finitepartinject-expr	cartesianprod-expr
natural-typer	positive-type	integer-type	real-type
character-type	digit-type	bool-type	pairfirst-expr
pairsecond-expr	emptyset-expr	generalunion-expr	generalintersect-expr
reldomain-expr	relrange-expr	cardinality-expr	successor-expr
minnumber-expr	maxnumber-expr	mapping-expr	numrange-expr
setunion-expr	setminus-expr	bagunion-expr	setintersect-expr
filtering-expr	composition-expr	backcompose-expr	overriding-expr
domainres-expr	rangeres-expr	antidomainres-expr	antirangeres-expr
powerset-expr	nonempty-expr	idrelation-expr	finiteset-expr
nefinite-expr	sequence-expr	neseq-expr	injseq-expr
z-bag-expr	inversion-expr	transclosure-expr	refclosure-expr
relimage-expr	op-name-expr	set-expr	set-display-expr
set-comp-expr	seq-expr	bag-expr	theta-expr
component-expr			

x followed by an assignment of the value of x to y. After execution of this method, both x and y would contain y's original value.

Finally, all specifications written in *Z* are implicitly conjuncted together, meaning that they all must be true following execution of a given method. However, it is also valid to string all predicates together by explicitly conjuncting them with "and" statements. This transformation system transforms conjunction to boolean "and" constructs, which is not correct if the intent was to explicitly conjunct two or more predicates. Therefore, each predicate should stand alone in the specification in order for the transforms to work properly.

While limitations exist, the system was validated by transforming an existing library of primitive specifications. Many of these specifications inherit attributes and methods from other specifications, further verifying that the transformation system correctly handles inheritance. For each correct and parsable specification found in the library, the system produces correct Ada 95 source code. This is in addition to Appendices A and B, which contain a specification and the Ada 95 source code produced from its transformation.

4.9 Implementation Wrap-Up

In this chapter it has been shown that, through the use of separate models for analysis and design and a series of transformations preparing each model for its conversion to the next phase, source code produced from formal specifications can be realized. The invariant constraints must be handled in the domain model before it can be transformed into a design model. Once the domain model is complete, a series of transformations produce a preliminary design model. Another set of transformations completes the design model and prepares it to produce output. Only then can the output grammars be used to produce compilable Ada code. For a complete specification in Z and its resulting source code produced from this transformation system, see Appendices A and B. Additional grammars could be produced that would generate executable code in C, C++ or a variety of other programming languages.

V. *Conclusions and Recommendations*

5.1 Accomplishments

This research effort proves that executable source code can be produced from formal specifications using a transformation system as the proof of concept. A sample formal specification involving inheritance is found in Appendix A with the source code produced from the specification found in Appendix B. The script used to compile the transformation system is in Appendix C. Several positive results came from the proof of concept that are worth mentioning here. First, by building on research already accomplished with *AFIT*tool, the transformations that handle the invariant constraints for primitive classes can now recognize and implement a correct solution for those constraints even when inheritance is involved. This was a significant contribution to the continuing effort of the overall *AFIT*tool system. Next, by studying the development of the Generic Object Model (GOM) developed by Sward [14], it was determined that with some modifications this model could provide a generic design to be used in transforming the domain model to a design model. The effort expended to locate, analyze and approve this model was far less than the effort to produce such a model from scratch. This savings allowed the research to concentrate a larger effort in other areas. By analyzing and implementing a series of transformations to transform the domain model structure into a design model structure, it was determined that most of the information necessary to complete the design model was readily available in the domain. However, when additional information was needed, the decision to build an interface to the human designer proved to be sufficient to capture the remaining information needed. Once the design model was complete the system was able to produce compilable Ada source code that met all of the assumptions listed in this research effort. The crux of this research has proved that the object and functional models found in the domain structure can be used to produce source code. There are, however, areas of the domain model that have still not been transformed and should be part of an additional research effort.

5.2 *Future Research*

The dynamic model portion of the domain model that deals with states, transitions and events was not addressed by this research due to time constraints. Some background work by Hartrum [2] has been done to validate an approach that a human developer might take. This process for transforming the state transition table into functional operations is currently a manual process, but is a fairly well-defined, step-by-step procedure. The desired methodology could emulate the manual process using a transformation. Since a design model has no concept of a state transition table, the purpose of this transformation would be to convert this table into operations that could become part of the design model. The approach to implementing this transformation begins with rearranging the state transition table, ordering it by event. Next, for each unique event in the state transition table, an operation is created called *Do_Event_Name* and the parameters of the event become input parameters of the new operation. Then for each row in the transition table corresponding to the event, a logical implication is built with the antecedent including the conjunction of the predicates from the current state and the guard conditions for the transition and a consequent including the conjunction of the predicates from the next state shown as post-conditions and the list of any actions and sends from the action column of the state transition table. This methodology does not address the possible logical simplification of the implication built from the state transition table. However, an approach for simplification could probably be developed. In addition, schemes for handling automatic transitions might be necessary.

Also, during the discussion of the user-defined types being transformed from the domain model to the design model, it was mentioned that some user-defined types have boundary conditions in the form of predicates. While these predicates were attached to the user-defined type in the design model, they were never transformed into any usable construct in the design. The goal of future research should be to convert these predicates into design model constructs and to build a functional method in the design where these constraints would be the statements of the method. The functional method would be considered a test to ensure that any value passed to the function meets the conditions of the type and as such could be named *Is_User_Defined_Type*. The function would return a

boolean true or false value indicating whether or not the value passed in met the conditions of the original predicates. It would then require some additional logic to place calls to this functional method throughout the design model where they would be needed. That would be the most complicated part of this problem.

In addition, this effort concentrated on converting certain predicates and expressions into program constructs in the design model. This approach does not provide a method for general algorithm design. Perhaps this tool could be interfaced to another tool capable of handling some level of algorithm design. The information produced by the second tool could possibly be used to produce more complete code.

Appendix A. Formal Specification for SubCounter Involving Inheritance

SubCounter Structure Definition

Object Name: SubCounter

Object Number: 98020X

Object Description: This is a counter with a dynamic and functional model. It includes a *limit*, maintains the maximum count achieved (*max_reached*), includes derived attribute *margin* to indicate the difference between the current count and the limit, and limits the value of *limit* to a constant *MAX_COUNT*. It reacts to events to set the up/down mode, to reset to zero, and to count in the direction indicated by the current mode. An alarm message is sent if an attempt is made to count beyond zero or *limit*.

NOTE: "count" is a reserved word in Refine! (won't work as an attribute).

Date: 09/16/98

History: 04/21/98: Original, test case for Trans96. 09/16/98: Changed syntax for COUNT_MODE for consistency with AFITtool changes.

Author: Hartrum

Superclass: None

Components: None

Context: None

Attributes:

thecount	integer	Current value of counter.
limit	integer	Max positive count.
margin	integer	Difference between thecount and limit.
max_reached	integer	highest count reached.
mode	COUNT_MODE	counting mode: up or down.
model	MODEL_TYPE	
model_year	YEAR	

Constraints:

thecount ≥ 0

thecount $\leq$ limit

max_reached $\leq$ limit

limit $\leq$ MAX_COUNT

Z Static Schema:

Counter

$$thecount : \mathcal{N}$$
$$limit : \mathcal{N}$$
$$thecount \leq limit$$

InitCounter

$$\Delta Counter$$
$$thecount' = 0$$
$$limit' = 100$$

$[MODEL\_TYPE, YEAR]$

$$MAX\_COUNT : \mathcal{N}$$

$$MAX\_COUNT = 1000$$

$$SUB\_YEAR : YEAR$$

$$SUB\_YEAR = 1000$$

$$SUB\_STATUS ::= on \mid off$$

$$ALL\_MODE ::= up \mid down \mid hold \mid off$$

$$COUNT\_MODE : P\, ALL\_MODE$$

$$\forall x : COUNT\_MODE \bullet ((x = up) \lor (x = down))$$

SubCounter

$$margin : \mathcal{N}$$

$$max\_reached : \mathcal{N}$$

$$mode : COUNT\_MODE$$

$$status : SUB\_STATUS$$

$$model : MODEL\_TYPE$$

$$model\_year : YEAR$$

Counter

$$model\_year \leq SUB\_YEAR$$

$$max\_reached \leq limit$$

$$limit \leq MAX\_COUNT$$

$$margin = limit - thecount$$

InitSubCounter

$\Delta SubCounter$

$max\_reached' = 0$

$mode' = up$

$model' = model\_A$

$model\_year' = 1996$

$margin' = 100$

SubCounter Functional Model

Object: *SubCounter*

Process Name: reset_count

Process Description: Set the count to zero.

Z Dynamic Schema:

ResetCount

$\Delta SubCounter$

$thecount' = 0$

Process Name: set_count

Process Description: Set the count to the specified value.

Z Dynamic Schema:

$SetCount$

$\Delta SubCounter$

$newcount? : \mathcal{N}$

$newcount? > 0$

$thecount' = newcount?$

Process Name: SetModeUp

Process Description: Set the counter to the up mode.

Z Dynamic Schema:

$SetModeUp$

$\Delta SubCounter$

$mode' = up$

Process Name: SetModeDown

Process Description: Set the counter to the down mode.

Z Dynamic Schema:

$SetModeDown$

$\Delta SubCounter$

$mode' = down$

Process Name: increment

Process Description: Add one to the count.

Z Dynamic Schema:

Increment

$\Delta SubCounter$

$thecount' = thecount + 1$

Process Name: decrement

Process Description: Subtract one from the count.

Z Dynamic Schema:

Decrement

$\Delta SubCounter$

$thecount' = thecount - 1$

Process Name: update_max

Process Description: If thecount exceeds max_reached, increment max_reached.

Z Dynamic Schema:

UpdateMax

$\Delta SubCounter$

$(thecount > max\_reached \Rightarrow max\_reached' = max\_reached + 1)$

Process Name: increment_count

Process Description: Increment thecount and update max_reached if exceeded. Decomposed into *Increment* followed by *UpdateMax*.

Z Dynamic Schema:

IncrementCount

$\Delta SubCounter$

SubCounter Dynamic Model

State Name: CountingUp

State Description: *thecount* is incremented in response to *DoCount.*

Z Static Schema:

CountingUp

$SubCounter$

$mode = up$

State Name: CountingDown

State Description: *thecount* is decremented in response to *DoCount.*

Z Static Schema:

CountingDown

$SubCounter$

$mode = down$

State Name: NotReset

State Description: A substate of the *CountingDown* state where the SubCounter has not been reset. Correct Z syntax won't pass uzed2dom.

Z Static Schema:

NotReset

$CountingDown$

$mode = down$

State Name: Reset

State Description: A substate of the *CountingDown* state where the SubCounter has been reset since entering the *CountingDown* state. Correct Z syntax won't pass uzed2dom.

Z Static Schema:

Reset

$CountingDown$

$mode = down$

Event Name: Count

Event Description: Advance *thecount* in the direction indicated by the current state.

Z Static Schema:

Count

$True$

Event Name: ReSet

Event Description: Reset *thecount* to zero.

Z Static Schema:

$$\begin{array}{|l} \textit{ReSet} \\ \hline \\ \textit{True} \\ \hline \end{array}$$

Event Name: Set

Event Description: Set *thecount* to value *newcount.*

Z Static Schema:

$$\begin{array}{|l} \textit{Set} \\ \hline newcount : \mathcal{N} \\ \hline newcount > 0 \\ \hline \end{array}$$

Event Name: SetMode

Event Description: Set the mode to the specified value.

Z Static Schema:

$$\begin{array}{|l} \textit{SetMode} \\ \hline newmode : COUNT\_MODE \\ \hline \textit{True} \\ \hline \end{array}$$

Event Name: Alarm

Event Description: The *Alarm* event is sent by the SubCounter.

Z Static Schema:

$$\begin{array}{|l} \textit{Alarm} \\ \hline \\ \textit{True} \\ \hline \end{array}$$

Event Name: Notice

Event Description: The *Notice* event is sent by the SubCounter.

Z Static Schema:

Notice

$True$

State Transition Table:

Current	Event	Guard	Next	Action	Send
CountingUp	ReSet		CountingUp	ResetCount	
CountingUp	Set		CountingUp	SetCount	
CountingDown	Set		CountingDown	SetCount	
CountingDown	ReSet		CountingDown	ResetCount	
CountingUp	SetMode	$newmode = down$	CountingDown	SetModeDown	
CountingUp	Count	$thecount < limit$	CountingUp	IncrementCount	
CountingUp	Count	$thecount = limit$	CountingUp		Alarm
CountingDown	SetMode	$newmode = up$	CountingUp	SetModeUp	
CountingDown	Count	$thecount > 0$	CountingDown	Decrement	
CountingDown	Count	$thecount = 0$	CountingDown		Alarm
NotReset	Count	$thecount > 0$	NotReset	Decrement	
NotReset	Reset		Reset		Notice
Reset	Count	$thecount > 0$	Reset	Decrement	

Appendix B. Source Code Produced for SubCounter

B.1 Counter Specification

The source code listed in this appendix was produced using an output grammar. It has been formatted for readability, but no text was modified.

```
--Written by ttankers on 1/19/1999
With udtypes; Use udtypes;

package COUNTER is
  Type COUNTER_REC  is tagged private;
  Type COUNTER_REFERENCE is access COUNTER_REC;

--methods
  procedure INITCOUNTER (Z_COUNTER : in out COUNTER_REC );

  procedure SET_THECOUNT (Z_COUNTER : in out COUNTER_REC;
                          X_THECOUNT : in natural );

  procedure SET_LIMIT (Z_COUNTER : in out COUNTER_REC;
                       X_LIMIT : in natural );

  function GET_THECOUNT (Z_COUNTER : in COUNTER_REC ) return natural;

  function GET_LIMIT (Z_COUNTER : in COUNTER_REC ) return natural;

Private
 Type  COUNTER_REC  is TAGGED
  RECORD
    THECOUNT :  natural;
    LIMIT :  natural;
  END RECORD;
end COUNTER;
```

B.2 Counter Body

```
--Written by ttankers on 1/19/1999
with Ada.Text_IO, Ada.Strings.Unbounded;
use Ada.Text_IO, Ada.Strings.Unbounded;

package body COUNTER is
 --methods

  procedure INITCOUNTER (Z_COUNTER : in out COUNTER_REC) is
  begin
    Z_COUNTER.THECOUNT := 0;
    Z_COUNTER.LIMIT := 100;
  end INITCOUNTER;

  procedure SET_THECOUNT (Z_COUNTER : in out COUNTER_REC;
                          X_THECOUNT : in natural) is
  begin
    if X_THECOUNT  <=  GET_LIMIT (Z_COUNTER) then
      Z_COUNTER.THECOUNT := X_THECOUNT;
    else
      Ada.Text_IO.Put_Line("Expression SET_THECOUNT in class COUNTER failed.");
    end if;
  end SET_THECOUNT;

  procedure SET_LIMIT (Z_COUNTER : in out COUNTER_REC;
                       X_LIMIT : in natural ) is
  begin
    if GET_THECOUNT (Z_COUNTER)  <=  X_LIMIT then
      Z_COUNTER.LIMIT := X_LIMIT;
    else
      Ada.Text_IO.Put_Line("Expression SET_LIMIT in class COUNTER failed.");
    end if;
  end SET_LIMIT;
```

```
  function GET_THECOUNT (Z_COUNTER : in COUNTER_REC ) return natural is
  begin
    return Z_COUNTER.THECOUNT;
  end GET_THECOUNT;

  function GET_LIMIT (Z_COUNTER : in COUNTER_REC ) return natural is
  begin
    return Z_COUNTER.LIMIT;
  end GET_LIMIT;

end COUNTER;
```

B.3 SubCounter Specification

```
--Written by ttankers on 1/19/1999
With udtypes, COUNTER;
Use udtypes, COUNTER;

package SUBCOUNTER is

  Type SUBCOUNTER_REC is new COUNTER_REC with private;
  Type SUBCOUNTER_REFERENCE is access SUBCOUNTER_REC;

  --superclasses: COUNTER
  --methods

  procedure INITSUBCOUNTER (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure RESETCOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure SETCOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                      NEWCOUNT : in natural);

  procedure SETMODEUP (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure SETMODEDOWN (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure INCREMENT (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure DECREMENT (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure UPDATEMAX (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure INCREMENTCOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC);

  procedure SET_MAX_REACHED (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                             X_MAX_REACHED : in natural);
```

```
procedure SET_MODE (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                    X_MODE : in COUNT_MODE);

procedure SET_STATUS (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                      X_STATUS : in SUB_STATUS);

procedure SET_MODEL (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                     X_MODEL : in MODEL_TYPE);

procedure SET_MODEL_YEAR (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                          X_MODEL_YEAR : in YEAR);

procedure SET_LIMIT (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                     X_LIMIT : in natural);

procedure SET_THECOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                        X_THECOUNT : in natural);

function GET_MARGIN (Z_SUBCOUNTER : in SUBCOUNTER_REC) return natural;

function GET_MAX_REACHED (Z_SUBCOUNTER : in SUBCOUNTER_REC) return natural;

function GET_MODE (Z_SUBCOUNTER : in SUBCOUNTER_REC) return COUNT_MODE;

function GET_STATUS (Z_SUBCOUNTER : in SUBCOUNTER_REC) return SUB_STATUS;

function GET_MODEL (Z_SUBCOUNTER : in SUBCOUNTER_REC) return MODEL_TYPE;

function GET_MODEL_YEAR (Z_SUBCOUNTER : in SUBCOUNTER_REC) return YEAR;

Private
 Type  SUBCOUNTER_REC  is new COUNTER_REC with
  RECORD
```

```
    MARGIN :  natural;
    MAX_REACHED :  natural;
    MODE :  COUNT_MODE;
    STATUS :  SUB_STATUS;
    MODEL :  MODEL_TYPE;
    MODEL_YEAR :  YEAR;
  END RECORD;
end SUBCOUNTER;
```

B.4 SubCounter Body

```
--Written by ttankers on 1/19/1999
with Ada.Text_IO, Ada.Strings.Unbounded;
use Ada.Text_IO, Ada.Strings.Unbounded;

package body SUBCOUNTER is

  --superclasses: COUNTER
  --local constants

    SUB_YEAR : constant YEAR  :=  1000;
    MAX_COUNT : constant natural  :=  1000;

--methods

  procedure INITSUBCOUNTER (Z_SUBCOUNTER : in out SUBCOUNTER_REC ) is
  begin
    INITCOUNTER ( COUNTER_REC( Z_SUBCOUNTER ));
    Z_SUBCOUNTER.MAX_REACHED := 0;
    Z_SUBCOUNTER.MODE := UP;
    Z_SUBCOUNTER.MODEL := to_unbounded_string("MODEL_A");
    Z_SUBCOUNTER.MODEL_YEAR := 1996;
    Z_SUBCOUNTER.MARGIN := 100;
  end INITSUBCOUNTER;

  procedure RESETCOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
  begin
    SET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER), 0);
  end RESETCOUNT;

  procedure SETCOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                      NEWCOUNT : in natural) is
  begin
    SET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER), NEWCOUNT);
```

```
end SETCOUNT;

procedure SETMODEUP (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
begin
  SET_MODE (Z_SUBCOUNTER, UP);
end SETMODEUP;

procedure SETMODEDOWN (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
begin
  SET_MODE (Z_SUBCOUNTER, DOWN);
end SETMODEDOWN;

procedure INCREMENT (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
begin
  SET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER),
                GET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER)) + 1);
end INCREMENT;

procedure DECREMENT (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
begin
  SET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER),
                GET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER)) - 1);
end DECREMENT;

procedure UPDATEMAX (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
begin
  if GET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER)) > GET_MAX_REACHED (Z_SUBCOUNTER) then
    SET_MAX_REACHED (Z_SUBCOUNTER, GET_MAX_REACHED (Z_SUBCOUNTER) + 1);
  else
    Ada.Text_IO.Put_Line("Expression UPDATEMAX in class SUBCOUNTER failed.");
  end if;
end UPDATEMAX;

procedure INCREMENTCOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC) is
```

```
begin
  null;
end INCREMENTCOUNT;

procedure SET_MAX_REACHED (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                           X_MAX_REACHED : in natural) is
begin
  if X_MAX_REACHED  <= GET_LIMIT (COUNTER_REC(Z_SUBCOUNTER)) then
    Z_SUBCOUNTER.MAX_REACHED := X_MAX_REACHED;
  else
    Ada.Text_IO.Put_Line("Expression SET_MAX_REACHED in class SUBCOUNTER failed.");
  end if;
end SET_MAX_REACHED;

procedure SET_MODE (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                    X_MODE : in COUNT_MODE) is
begin
  Z_SUBCOUNTER.MODE := X_MODE;
end SET_MODE;

procedure SET_STATUS (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                      X_STATUS : in SUB_STATUS) is
begin
  Z_SUBCOUNTER.STATUS := X_STATUS;
end SET_STATUS;

procedure SET_MODEL (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                     X_MODEL : in MODEL_TYPE) is
begin
  Z_SUBCOUNTER.MODEL := X_MODEL;
end SET_MODEL;

procedure SET_MODEL_YEAR (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                          X_MODEL_YEAR : in YEAR) is
```

```
begin
  if X_MODEL_YEAR  <=  SUB_YEAR then
    Z_SUBCOUNTER.MODEL_YEAR := X_MODEL_YEAR;
  else
    Ada.Text_IO.Put_Line("Expression SET_MODEL_YEAR in class SUBCOUNTER failed.");
  end if;
end SET_MODEL_YEAR;

procedure SET_LIMIT (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                     X_LIMIT : in natural) is
begin
  if ((GET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER)) <= X_LIMIT) and
      (GET_MAX_REACHED (Z_SUBCOUNTER) <= X_LIMIT)) and
      (X_LIMIT <= MAX_COUNT) then
    SET_LIMIT ( COUNTER_REC( Z_SUBCOUNTER ), X_LIMIT);
    Z_SUBCOUNTER.MARGIN := GET_LIMIT (COUNTER_REC(Z_SUBCOUNTER))  -
                           GET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER));
  else
    Ada.Text_IO.Put_Line("Expression SET_LIMIT in class SUBCOUNTER failed.");
  end if;
end SET_LIMIT;

procedure SET_THECOUNT (Z_SUBCOUNTER : in out SUBCOUNTER_REC;
                        X_THECOUNT : in natural) is
begin
  if X_THECOUNT <= GET_LIMIT (COUNTER_REC(Z_SUBCOUNTER)) then
    SET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER), X_THECOUNT);
    Z_SUBCOUNTER.MARGIN := GET_LIMIT (COUNTER_REC(Z_SUBCOUNTER))  -
                           GET_THECOUNT (COUNTER_REC(Z_SUBCOUNTER));
  else
    Ada.Text_IO.Put_Line("Expression SET_THECOUNT in class SUBCOUNTER failed.");
  end if;
end SET_THECOUNT;
```

```
  function GET_MARGIN (Z_SUBCOUNTER : in SUBCOUNTER_REC) return natural is
  begin
    return Z_SUBCOUNTER.MARGIN;
  end GET_MARGIN;

  function GET_MAX_REACHED (Z_SUBCOUNTER : in SUBCOUNTER_REC) return natural is
  begin
    return Z_SUBCOUNTER.MAX_REACHED;
  end GET_MAX_REACHED;

  function GET_MODE (Z_SUBCOUNTER : in SUBCOUNTER_REC) return COUNT_MODE is
  begin
    return Z_SUBCOUNTER.MODE;
  end GET_MODE;

  function GET_STATUS (Z_SUBCOUNTER : in SUBCOUNTER_REC) return SUB_STATUS is
  begin
    return Z_SUBCOUNTER.STATUS;
  end GET_STATUS;

  function GET_MODEL (Z_SUBCOUNTER : in SUBCOUNTER_REC) return MODEL_TYPE is
  begin
    return Z_SUBCOUNTER.MODEL;
  end GET_MODEL;

  function GET_MODEL_YEAR (Z_SUBCOUNTER : in SUBCOUNTER_REC) return YEAR is
  begin
    return Z_SUBCOUNTER.MODEL_YEAR;
  end GET_MODEL_YEAR;

end SUBCOUNTER;
```

B.5 User Defined Types Specification

```
With Ada.Strings.Unbounded;

Package UDTYPES is

  Type SUB_STATUS is (on, off);
  Type ALL_MODE is (up, down, hold, off);
  SubType MODEL_TYPE is Ada.Strings.unbounded.unbounded_string;
  SubType YEAR is natural;
  SubType COUNT_MODE is ALL_MODE;

end UDTYPES;
```

Appendix C. System Compilation Sequence

```
; Lisp file used to compile design model, transforms and grammars
(compile-and-load-file "gom-model")
(compile-and-load-file "dom-2-gom-xforms")
(compile-and-load-file "ada_spec")
(compile-and-load-file "ada_body")
(compile-and-load-file "gom-utils")
(compile-and-load-file "gom-2-gom-xforms")
(compile-and-load-file "gom-utils2")
(compile-and-load-file "gomtool")
```

Bibliography

1. Anderson, Gary L. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model.* MS thesis, AFIT/GCS/ENG/99M-01, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, Mar 1999.
2. Hartrum, Thomas C. "Object Oriented Design." Unpublished, Sep 1997.
3. Hartrum, Thomas C. "An Object Oriented Formal Transformation System for Primitive Object Classes." Unpublished, Mar 1998.
4. Kissack, John. *Transforming Aggregate Object-Oriented Formal Specifications to Code.* MS thesis, AFIT/GCS/ENG/99M-09, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, Mar 1999.
5. Lowry, Michael and Raul Doran. *The Handbook of Artificial Intelligence, Volume IV.*, chapter 20. Reading, MA: Addison Wesley, 1989.
6. Martin, John C. *Introduction to Languages and the Theory of Computation.* Boston, MA: McGraw-Hill, 1997.
7. Reasoning Systems Inc. *Dialect User's Guide*, 1990.
8. Reasoning Systems Inc. *REFINE User's Guide*, 1995.
9. Reasoning Systems Inc. *REFINE/Ada User's Guide Version 2.0*, 1998.
10. Reasoning Systems Inc. *REFINE/C User's Guide Version 1.2*, 1998.
11. Rumbaugh, James and others. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.
12. Schorsch, Thomas M. "Transformation of Generic Object Model Abstract Sytax Tree to Ada Abstract Syntax Tree.." Unpublished - Not Dated.
13. Spivey, J.M. *The Z Notation: A Reference Manual.* Technical Report OX1 4EW, Oxford, England: Oriel College, 1998.
14. Sward, Ricky E. *Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code.* PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, Sep 1997. AFIT/DS/ENG/97-04.

Vita

Travis W. Tankersley was born on 24 September 1966 in Tullahoma, Tennessee. He graduated from high school in Lynchburg, Tennessee in 1984. He then attended Motlow State Community College on a 2-year scholarship. He married Michelle Miller in 1987 in Winchester, Tennessee. From 1987 to 1991 he worked as a draftsman in Nashville, Tennessee. In 1991, he enlisted in the Air Force. His first assignment was as an operating systems programmer for the AWACS program at Tinker AFB, Oklahoma. His son Trent was born in 1993 during his AWACS tour. He completed his undergraduate degree at Park College in 1994. Following graduation, he was accepted to Officer Training School and received his commission in 1995. In Aug, 1997 he reported to the Graduate School of Engineering, Air Force Institute of Technology.

Permanent address: 2400 Rona Village Blvd.
Fairborn, OH 45324

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	8 Mar 1999	Master's Thesis

4. TITLE AND SUBTITLE
Generating Executable Code from Formal Specifications of Primitive Objects

5. FUNDING NUMBERS

6. AUTHOR(S)
Travis W. Tankersley, 1st Lieutenant, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/99M-19

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/IFTD
Mr Roy F. Stratton, Jr.
525 Brooks Rd.
Rome, NY 13441-4505
(315) 330-3004

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Maj Robert P. Graham, Jr.
(937) 255-3636 x4595
Robert.Graham@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The concept of developing a model for producing compilable and executable code from formal software specifications has long been a goal of software engineers. Previous research at the Air Force Institute of Technology (AFIT) has been focused on specification and domain analysis. An analysis model is populated using specifications written in Z. Then, a set of preliminary design transforms refines the specification in the analysis model. This research bridges the gap between analysis and design, allowing source code to be produced from formal specifications of primitive objects using transformational programming. The contribution of this thesis is to transform the analysis model for primitive objects into a design model representing the primitive objects, and to produce compilable and executable source code in Ada 95 from the resulting design model.

14. SUBJECT TERMS
Transformation, code generation, *Z* specifications, domain analysis, design model, Ada.

15. NUMBER OF PAGES
102

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL